Extracted from:

Seven Mobile Apps in Seven Weeks

Native Apps, Multiple Platforms

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Seven Mobile Apps in Seven Weeks

Native Apps, Multiple Platforms

Tony Hillerson

Series editor: *Bruce A. Tate*

Development editor: *Jacquelyn Carter*

Seven Mobile Apps in Seven Weeks

Native Apps, Multiple Platforms

Tony Hillerson

The Pragmatic Bookshelf

Raleigh, North Carolina

The team that produced this book includes:

Jacquelyn Carter (editor)
Potomac Indexing, LLC (index)
Candace Cunningham, Molly McBeath (copyedit)
Gilson Graphics (layout)
Janet Furlow (producer)

Printed in the United States of America.
ISBN-13: 978-1-68050-148-3
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2016

Time and time again in my journey as a mobile developer, clients have asked about cross-platform tools. “We want an iOS *and* an Android app; can we do that with less time and money with *tool x*?” they ask. I have to confess to feeling irritated by this sort of question. For many reasons, some valid, some perhaps elitist, I have a gut feeling that mobile development should be done with the official tools.

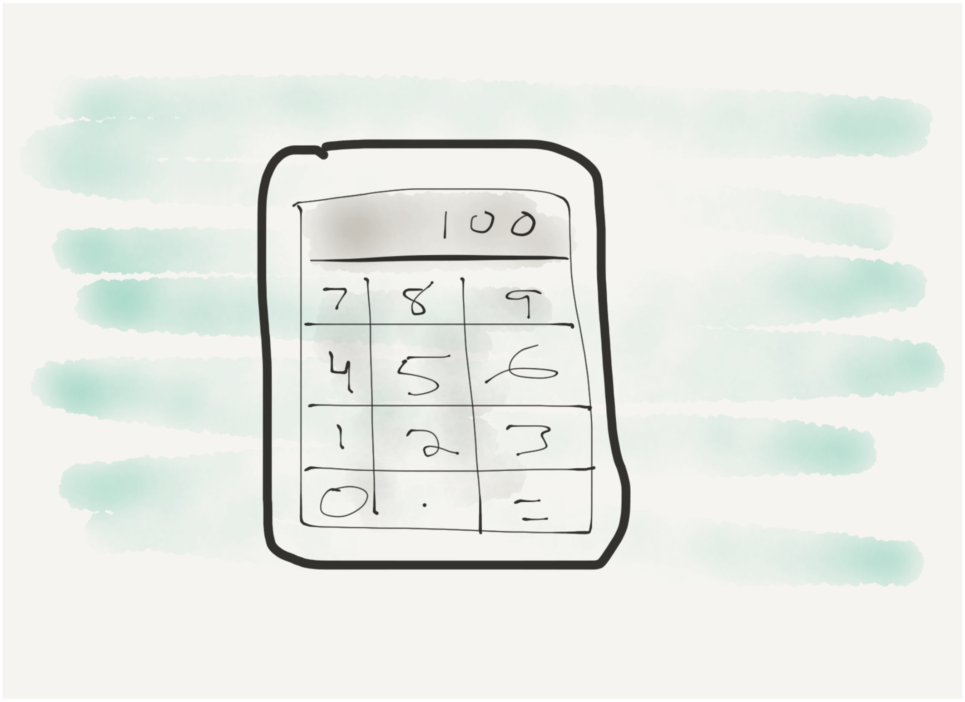
On the other hand, some of my reluctance to use anything but the platform-native frameworks is due to the number of bad apps out there built with shortcut solutions, especially those that try to use a web view as the shortcut. Then again, developing apps can’t happen in a vacuum. If there’s no business reason to build apps, developers will be out of a job. It’s our responsibility as developers to guide the stakeholders in the right direction, but also to be able to see the development process from the stakeholders’ point of view. Can we build their apps with less time and money? Can we make quality apps, not shortcuts? And can we keep the codebase maintainable?

Xamarin was the first tool that caused me to rethink my assumption that all cross-platform tools were intrinsically bad. With Xamarin, you write *native* apps, not web-view apps, using platform-native APIs for iOS, Android, or Windows. You can share code between all of these. You write the apps in C#, and they run natively on the Mono .NET runtime,[1] so you have access to the .NET platform as well as the host platform. That’s a powerful combination.

All things being equal, I still prefer development with the official tools; I *like* the official tools. But after using it, I believe that in a business setting, with limited time and resources, Xamarin is a strong contender as a powerful tool for efficiently building apps for multiple platforms.

1. http://www.mono-project.com/

A Calculator App



This week we'll build a calculator app. It's going to have a UI with a display and a grid of buttons and will run on both iOS and Android. We'll use shared code for the calculator "engine" so that we have to write that only once. The code will be well tested, both in business logic and on the UI. Finally, we'll add a feature allowing the user to convert a value in the calculator as if it were a currency value, from one currency to another.

On Day 1 we'll build out the iOS and Android UIs and see how platform-specific development works in Xamarin. Then, on Day 2, we'll test and build a calculator model, wire it up to the UI, and then test the UI. On Day 3 we'll use Xamarin.Forms to rewrite a cross-platform UI, cutting down the platform-specific code to almost zero. We'll finish out Day 3 by consuming data from our currency-conversion API. Let's get started!

Day 1: Adding Up the Platforms

Or, Building an iOS and Android View

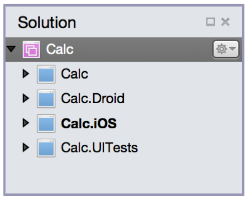
Today we'll dive in and build a Xamarin *solution*, or set of projects, which will have both an Android and an iOS UI for a calculator. Most of the code we write will use native APIs and widgets on those platforms, just written in C# instead of native code.

Creating and Setting Up a Solution

The first step is to use Xamarin Studio to create a new solution. We want to build this app for both iOS and Android, so we need to choose to build a cross-platform app. We'll start with the native single-view app this time. On Day 3 we'll rebuild the UI using Xamarin.Forms, so don't choose that option at this time. We also want to have unit tests generated, so make sure you check the box labeled Add an Automated UI Test Project.

The app name should be set to Calc, and the identifier should be set to your reverse domain name. The solution will contain an iOS-native project, an Android-native project, and a shared-code project. There are two options for how to configure that project, and we want it to be a *Portable Class Library*, or PCL. Once the solution is created, you'll see the project layout in the *Solution pane*.

The solution contains four projects: Calc.Droid for Android-specific code, Calc.iOS for iOS-specific code, Calc for shared code, and, finally, Calc.UITests for all the testing needs of the solution. Let's look at some of the generated code in the native projects. First, we'll look at the main Android activity.



Xamarin/xamarin_01_02_cleanup_generated_comments/Calc/Droid/MainActivity.cs

```
namespace Calc.Droid {

  [Activity(Label = "Calc.Droid", MainLauncher = true, Icon = "@drawable/icon")]
  public class MainActivity : Activity {
    int count = 1;

    protected override void OnCreate(Bundle bundle) {
      base.OnCreate(bundle);
      SetContentView(Resource.Layout.Main);
      Button button = FindViewById<Button>(Resource.Id.myButton);

      button.Click += delegate {
        button.Text = string.Format("{0} clicks!", count++);
      };
    }
  }
}
```

If you've gone through the Android chapter or are familiar with Android programming, you can pick out the Android APIs right away, just transcribed into C#. With Xamarin, access to native APIs on the underlying platform is gained through the C# versions of those APIs. So, for instance, in this example Android's Java android.app.Activity becomes Xamarin's C# Android.App.Activity and the onCreate life-cycle method is OnCreate, with a capital O.

You can see that some extra configuration can be done using C#'s *attributes*, in this example the code in brackets above the MainActivity class definition. These attributes are in many ways analogous to Java attributes.

The iOS-generated code is similarly familiar to iOS developers, just in C# instead of Objective-C. Currently, Xamarin mirrors the Objective-C APIs, not the Swift versions. Have a look at the generated AppDelegate, shown here.

Xamarin/xamarin_01_02_cleanup_generated_comments/Calc/iOS/AppDelegate.cs

```
namespace Calc.iOS {

  [Register("AppDelegate")]
  public class AppDelegate : UIApplicationDelegate {
    public override UIWindow Window { get; set; }

    public override bool FinishedLaunching(
      UIApplication application, NSDictionary launchOptions) {
      return true;
    }
  }
}
```

In Objective-C the *application delegate* conforms to the protocol UIApplicationDelegate; in Xamarin it extends the C# class UIApplicationDelegate and is registered with the Register attribute.

Overridden method names try to match the somewhat exotic Objective-C style as best as they can; for instance, in this example FinishedLaunching is analogous to application:didFinishLaunchingWithOptions, and it will be called when the application is launched and running, just as with any iOS app.

As we get further into the code, we'll see what more and more of the C# syntax means, so don't worry if you don't understand everything in even these simple generated-code samples. Let's get started building out the app now, beginning with the iOS view.

Building an iOS Calculator View

When working with Apple's official tooling, iOS views are generally built with interface bundles and storyboards. These files are edited in Xcode's visual Interface Builder editor, and the file formats aren't meant to be hand-edited.

Xamarin's strategy for dealing with this fact of iOS development has been to build visual tools that generally copy Interface Builder feature for feature and that operate on the same files that Xcode would. In fact, it's possible to continue to use Xcode to edit all the view files, if you like. We'll do everything using Xamarin Studio. Let's get started by adding a text field to act as the calculator display.

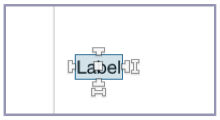
Adding the Calculator Display

Get started by opening up Calc.iOS/Main.storyboard. You should see a simulation of an iOS screen in the main view. This is where we can visually build the app's view. Make sure the Toolbox and Properties Pads are open by selecting View > Pads > Toolbox and View > Pads > Properties.

From the Toolbox, drag a label control out onto to the screen and drop it in the top-left side. As you get close to the left side of the screen, a guide line should pop up around 20 pixels from the edge. It's not important to be exact at this point, however; we'll adjust the label's positioning next.

Adding Auto Layout Constraints for the Label

With the label selected, click it once more and you should see the handles around the selection box change slightly, replacing the circles with squares, as you can see here.

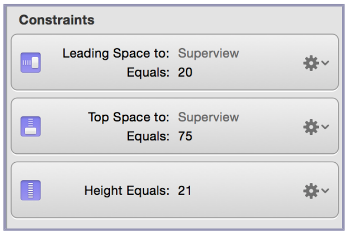


Each of these handles represents an *Auto Layout constraint.* Auto Layout is Apple's flexible layout system. Developers can add constraints to views to tell the system how to resize and position elements on the screen as the geometry of the view changes. Using Auto Layout can help you build views that can work on an iPhone and an iPad, by having widgets resize themselves or perhaps follow completely different rules based on the current device. When a widget is in the mode where it has square handles, you can drag the handles to create constraints relative to different parts of the UI, and you can also edit constraints with other tools in Xamarin Studio, which we'll look at next.

Auto Layout is very powerful, but we can't cover a lot of it today. You can find a more in-depth overview in *Introducing Auto Layout*, on page ?. All we'll do right now is set up constraints to have the label position itself and resize to keep to the full width of the screen.

Configuring Constraints and Other Properties

As a place to start, we can have Xamarin Studio add some suggested constraints. With the label still selected, go to the top of the screen and find the constraint controls. Click the green + symbol, and some constraints should be created based on where the label is positioned. To review the constraints, go to the Properties Pad and select the Layout tab. You should see something like this, even if the values are not exactly the same.



Constraints

Leading Space to: Superview
Equals: 20

Top Space to: Superview
Equals: 75

Height Equals: 21

Change the values for the constraints to those listed here, adding any that you need.

- Leading Space: 20
- Top Space: 20
- Trailing Space: 20
- Height: 60

Leading space is the space on the left side of the widget and trailing space is on the right. Finally, to get the rest of the settings right for the label, go to the Widget tab of the Properties Pad and change the following properties:

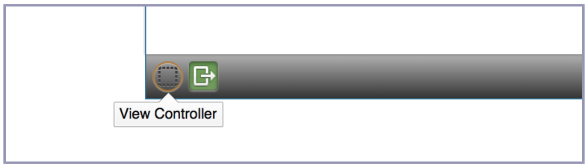
- Label Text: 0
- Alignment: Right
- Font: System 48 pt
- Background Color: Light Gray Color

Now, with the display configured, let's make a quick change to be able to run the app and see the results in the iOS simulator.

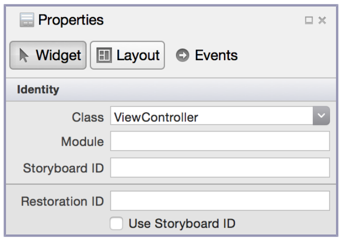
Viewing the Display

The code that Xamarin Studio generated for the iOS and Android projects had some sample code that we don't want to run right now. Instead of editing the code yet, we can change the storyboard to use a generic view controller instead of the generated one.

In the storyboard, at the bottom of the simulated iOS screen, select the left-hand button.



Then, in the Properties Pad, find the Class drop-down, which should contain the property ViewController. This is the generated controller class that we looked at first today.



Identity

Class ViewController

Module

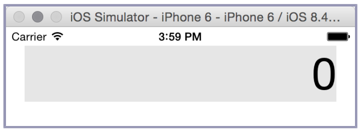
Storyboard ID

Restoration ID

Use Storyboard ID

Change this value to UIViewController, which is the base view controller class. It isn't a good idea to ship like this, but this will allow us to simply run the app and see the contents of the storyboard.

In the Solution Pad, right-click the Calc.iOS project and select Run Item. This will compile the app and deploy it to the iOS simulator. You should see something that looks remarkably like what we've already seen in Xamarin Studio in the storyboard.



So far, so good. Now let's move on to creating the calculator buttons. The buttons will be cells in a *UICollectionView*, which is iOS's widget for a group of related views. It has a flexible layout system that will allow us to lay the buttons out in a grid.

Adding the Calculator Buttons

Back in Xamarin Studio, go to the Toolbox Pad and drag a collection view to the storyboard view. Set up constraints as in the list that follows. This time, add the constraints using the constraint handles. Wherever a constraint from the collection view is relative to the label, drag the appropriate handle to the label.

- Top Space to Label Equals 0
- Align Leading to Label Equals 0
- Align Trailing to Label Equals 0
- Bottom Space to View Controller Top Layout Guide Equals 20

Next, edit the properties of the collection view, setting the background color to white and its name to ButtonCollectionView. Then set the label's name to DisplayLabel. These will be the variable names we'll use to reference these widgets in the view controller code.

Setting the name of the collection view and label in the storyboard causes a bit of magic to happen behind the scenes that is worth explaining here. If you look at ViewController in the Solution Pad, you'll see that you can twirl down the item in the tree, revealing a file called ViewController.designer.cs. This is a generated class that gets mixed in to ViewController and provides instance variables named according to the name properties in the storyboard.

Speaking of the view controller, go and change the storyboard's View Controller property back to the ViewController class. We'll be doing the rest of the work in that class instead of in the storyboard.

Creating a Collection View Cell for Buttons

We could create a collection view cell in the storyboard, but we can also do it in code. Here's the class for the calculator buttons, which extends 'UICollectionViewcell'.

Xamarin/xamarin_01_04_ios_calculator_collection_view/Calc/iOS/ViewController.cs

```
public class ButtonCell : UICollectionViewCell {

  public static NSString cellId = new NSString("ButtonCell");

  UILabel label;

  [Export("initWithFrame:")]
  public ButtonCell(RectangleF frame) : base(frame) {
    ContentView.Layer.BorderColor = UIColor.LightGray.CGColor;
    ContentView.Layer.BorderWidth = 0.5f;

    label = new UILabel();
    label.TextColor = UIColor.Black;
    label.TextAlignment = UITextAlignment.Center;
    label.Frame = new CGRect(10, 10, 30, 30);
    ContentView.AddSubview(label);
  }

  public void SetTitle(string title) {
    label.Text = title;
  }
}
```

The responsibility of this class is to configure the label and overall view of the button, as well as to expose a method, SetTitle, to change the button's label. One interesting Xamarin-specific item of interest is the attribute decorating the ButtonCell constructor.

The iOS constructor method for a collection view cell is called initWithFrame:, but C# requires that a constructor have the same name as the class, ButtonCell in this case. This Export attribute helps bridge that gap, telling Xamarin to associate the two methods.

With the button cell defined, now let's look at controlling the collection view with ViewController.