Write Once

Follow-along workbook

Getting Started

Xamarin relies on the platform SDKs from Apple, Google and Microsoft, which each have their own specific requirements. Complete details are available from Microsoft at:

https://docs.microsoft.com/en-us/xamarin/cross-platform/get-started/requirements

In order to use Xamarin, ensure your workstation meets these minimum requirements.

	macOS	**Windows**
Development Environment	Visual Studio for Mac	Visual Studio
Xamarin.iOS	Yes	Yes (with attached Mac)
Xamarin.Andriod	Yes	Yes
Xamarin.Forms	iOS & Android Only	Android, Windows/UWP (iOS with attached Mac)
Xamarin.Mac	Yes	No

Note: to develop for iOS on a Windows computer, there must be a Mac computer accessible on the network for remote compilation with XCode and debugging. Common scenarios are a Windows VM on a Mac or a Windows workstation connecting to a Mac. More details at: https://docs.microsoft.com/en-us/xamarin/ios/get-started/installation/windows/connecting-to-mac/.

GitHub Repo

Branches representing the various stages of this workshop have been created in GitHub at the following location:

https://github.com/jansmann/2019UAITSummit

With a few notable exceptions, most of this workbook can be completed by simply following the cookbook. However, feel free to start with one of the branches and then add to it as you go. Or, use the branches to clear out work that didn't quite go as planned so you can start clean on the next step. We won't judge.

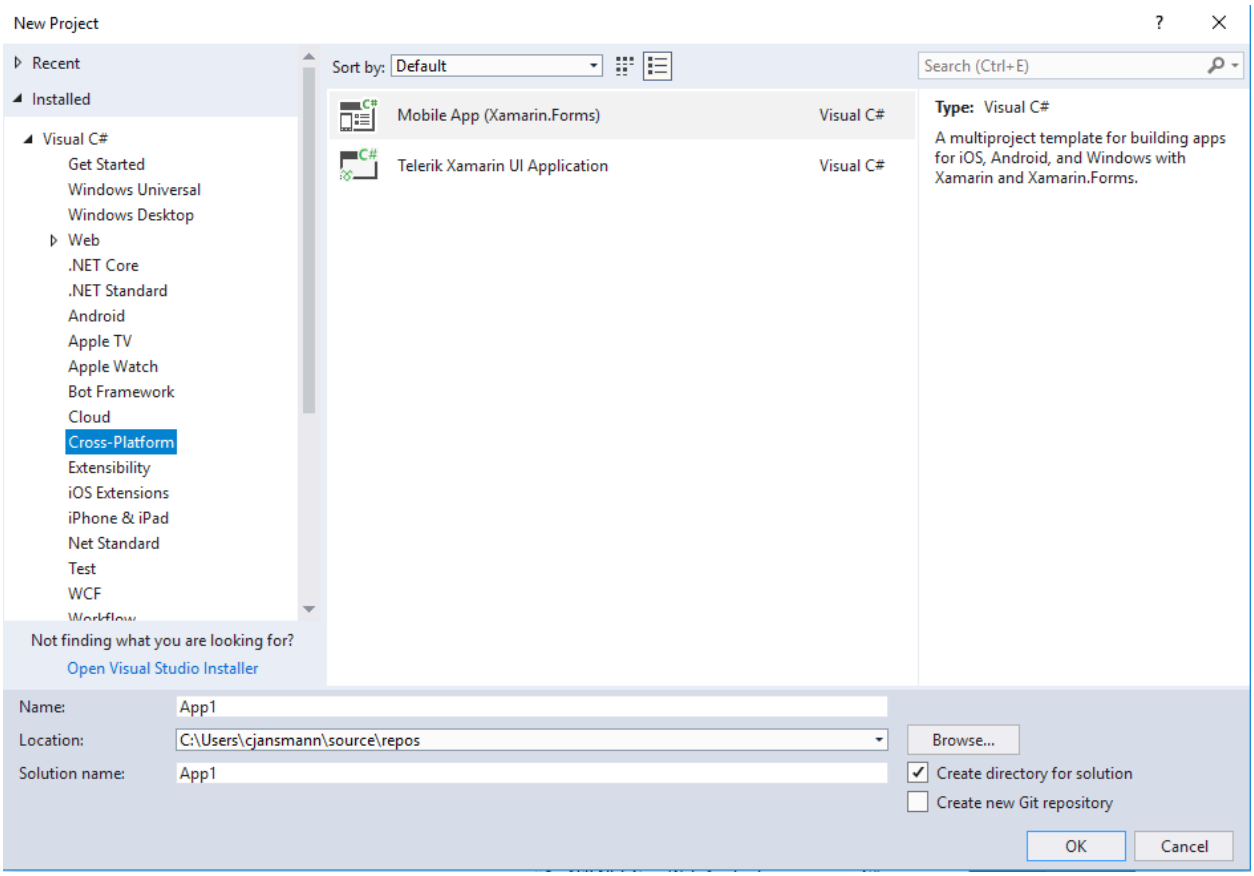
Most of all, please ask questions as you work.

Exercise One: Create the Project

Branch: 01-InitialConfig

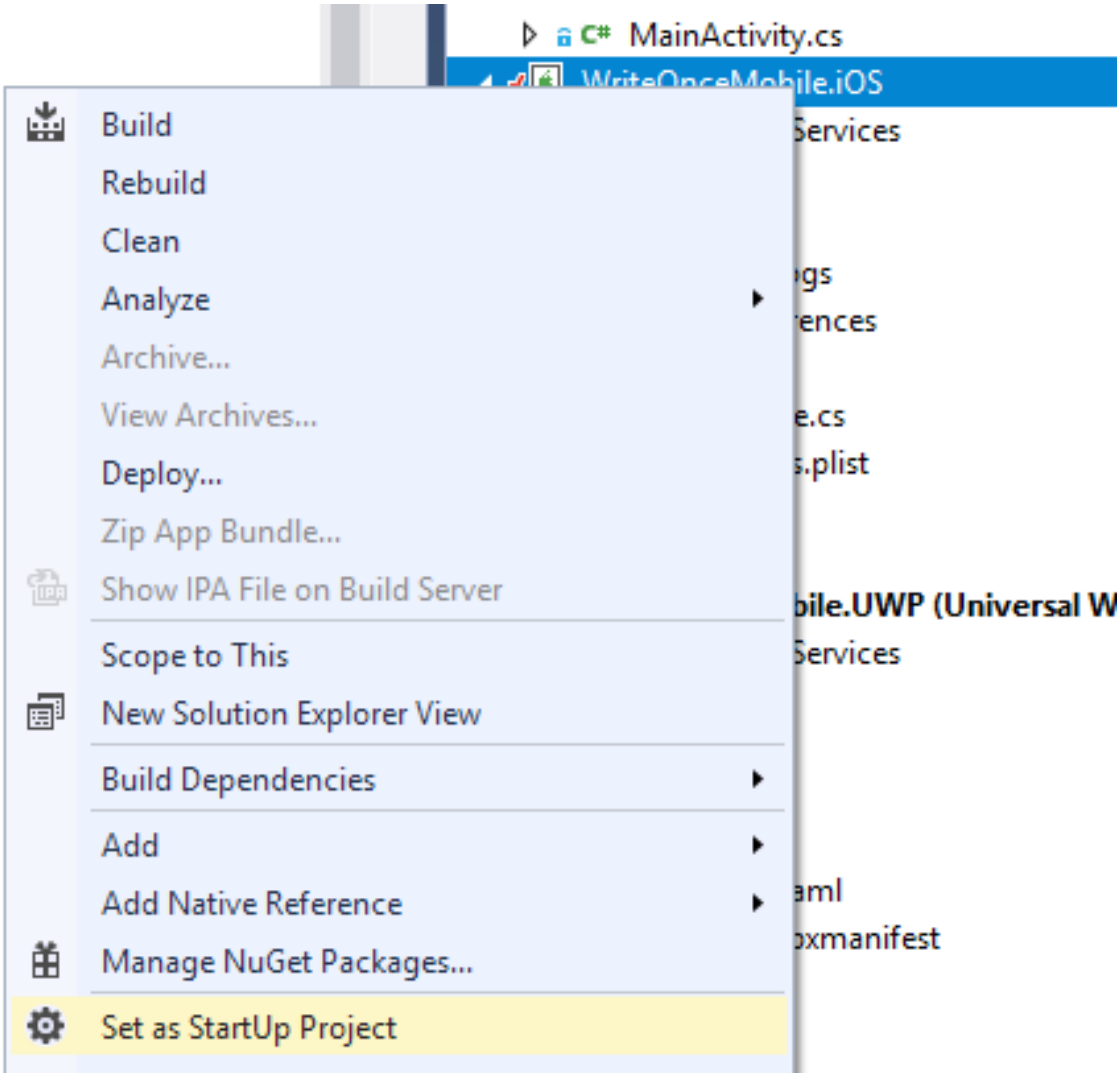
Windows

1. Load Visual Studio.
2. **File -> New Project**
3. Expand the **Installed -> Visual C#** node and select *Cross Platform* from the list.

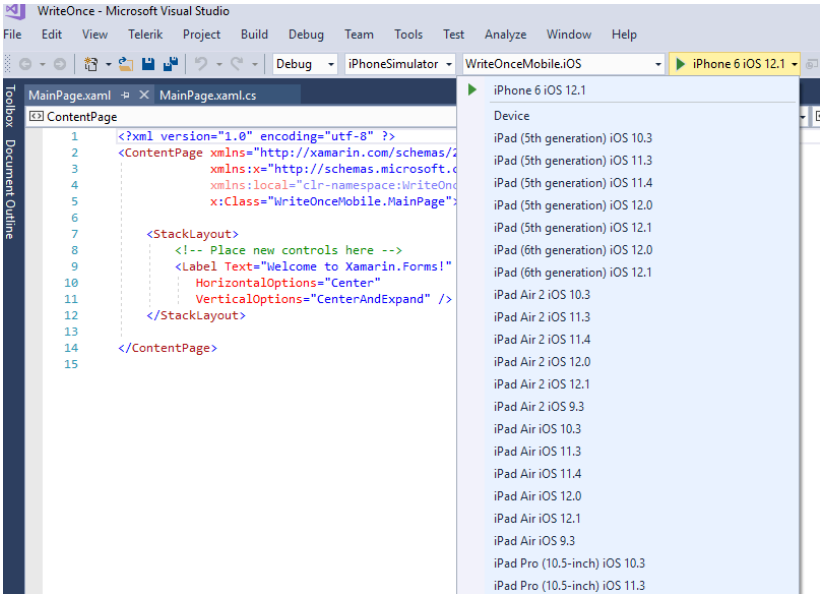


4. Select *Mobile App (Xamarin Forms)* and give your project a name.
    a. Project Name: WriteOnceMobile
    b. Solution Name: WriteOnce
5. Make sure the location you specify is one that you have full rights to.
6. Four projects will appear in your Solution Explorer
    a. WrtieOnceMobile – this is the shared project where most of our code will live
    b. WriteOnceMobile.Android – the Android SDK-based product
    c. WriteOnceMobile.iOS – the iOS-based product
    d. WriteOnceMobile.UWP – the Windows desktop and mobile product

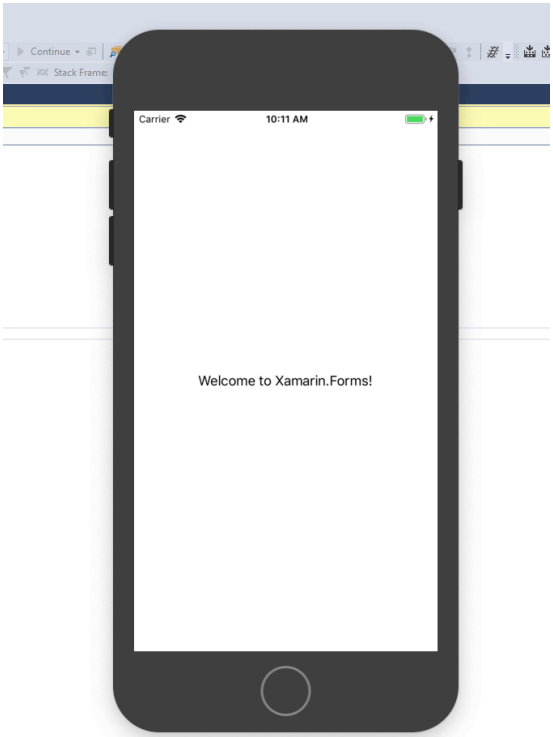
7. Assuming the Mac has been connected properly, right click on the *WriteOnceMobile.iOS* project and set it as your startup project.



8. Adjust your Build Settings on the toolbar to specify the kind of iOS device you want to test on:



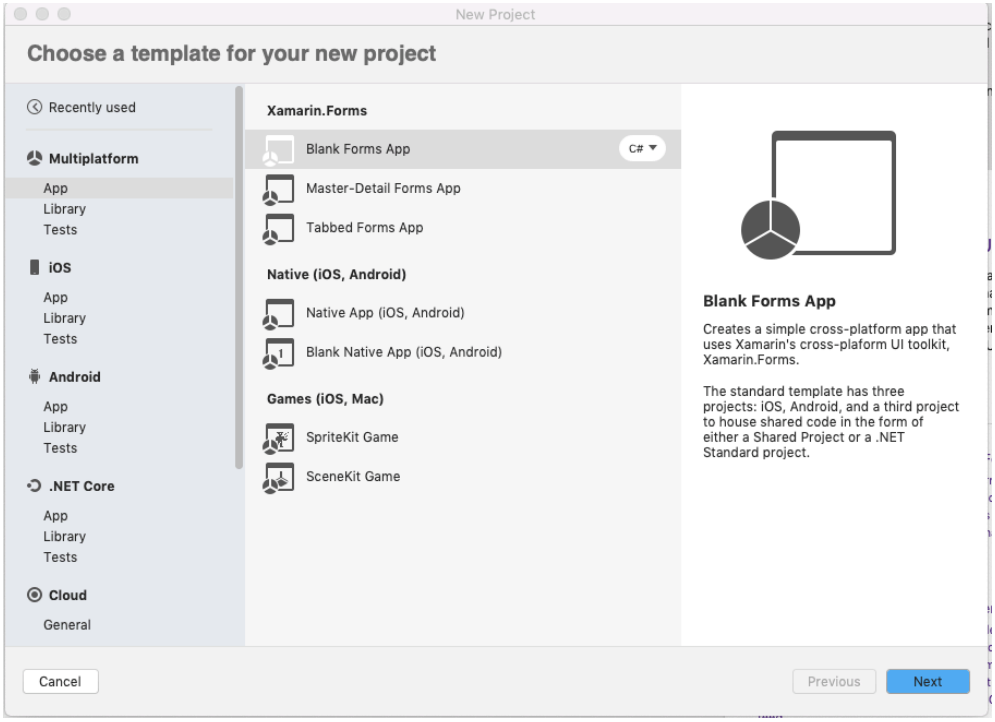
9. Try to build and run your project! Click the green triangle button next to your selected device or select *Debug -> Start Debugging* from the menu.
10. The iOS Simulator should fire up and land here:



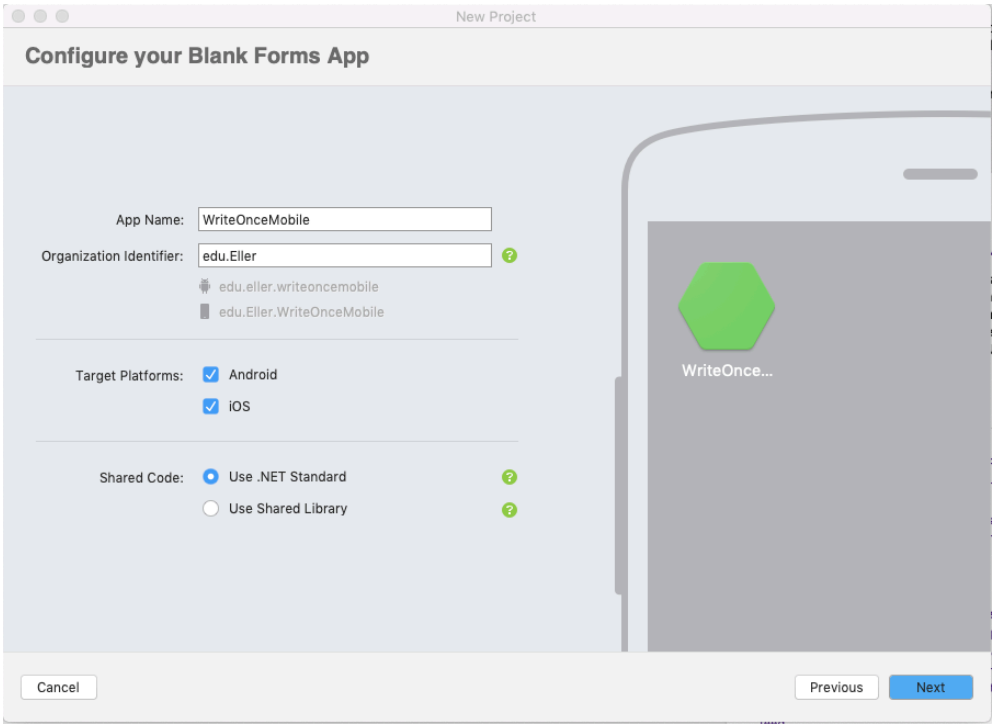
Mac OS

1. Launch Visual Studio for the Mac.
2. Click on the *New Project* button.

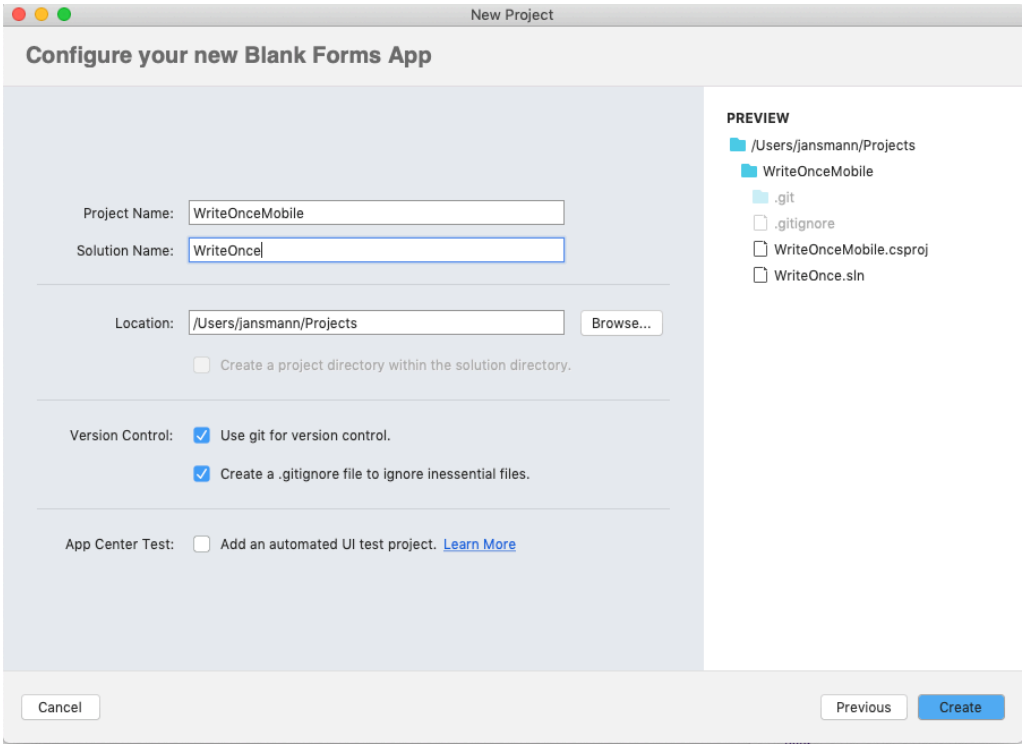
3. Select **Multiplatform -> App** from the side menu, and then *Blank Forms App*:



4. Give your app a name and customize the company name, and make sure *.NET Standard* is selected as your shared code option. **Note: Windows UWP apps will not be available as an option when starting the project in this manner.**

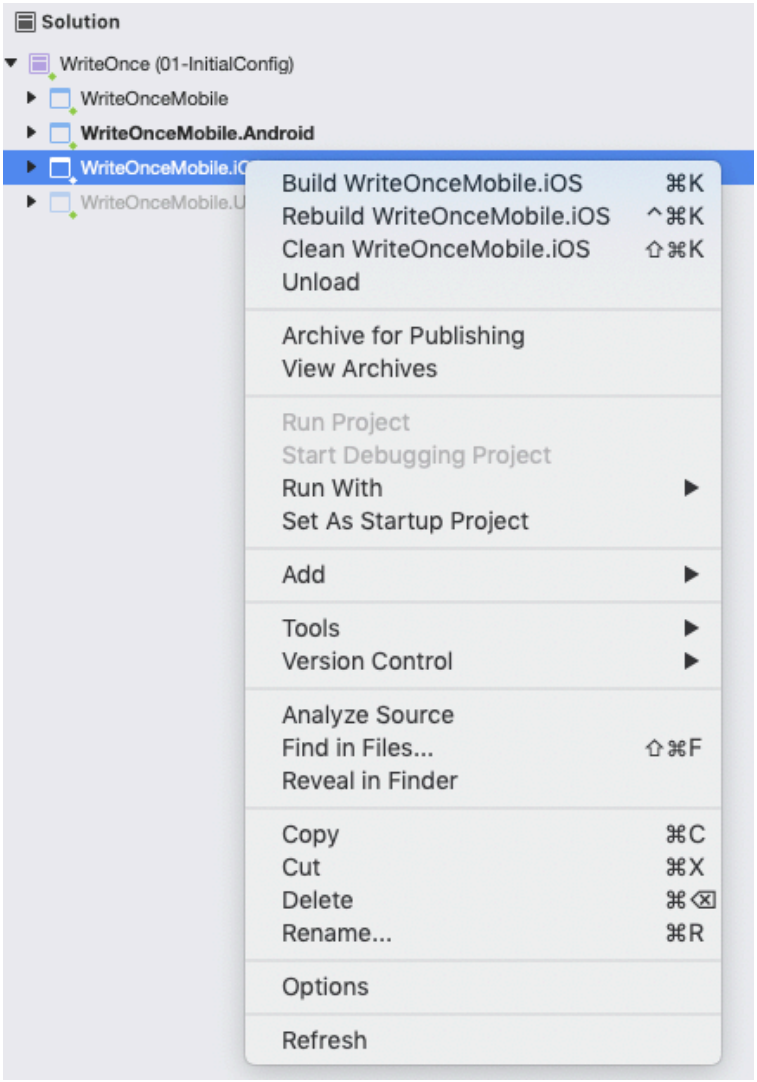


5. Make sure you have specified a solution name, select a place to store your project, and select the appropriate version control options.

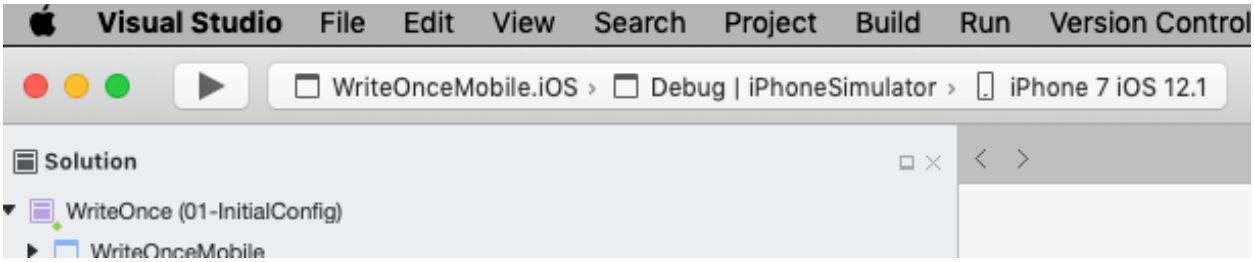


6. Four projects will appear in your Solution Explorer
    a. WrtieOnceMobile – this is the shared project where most of our code will live
    b. WriteOnceMobile.Android – the Android SDK-based product
    c. WriteOnceMobile.iOS – the iOS-based product
    d. WriteOnceMobile.UWP – the Windows desktop and mobile product. (On the Mac, this will appear as disabled)

7. Right click on WriteOnceMobile.iOS and set it as your startup project.

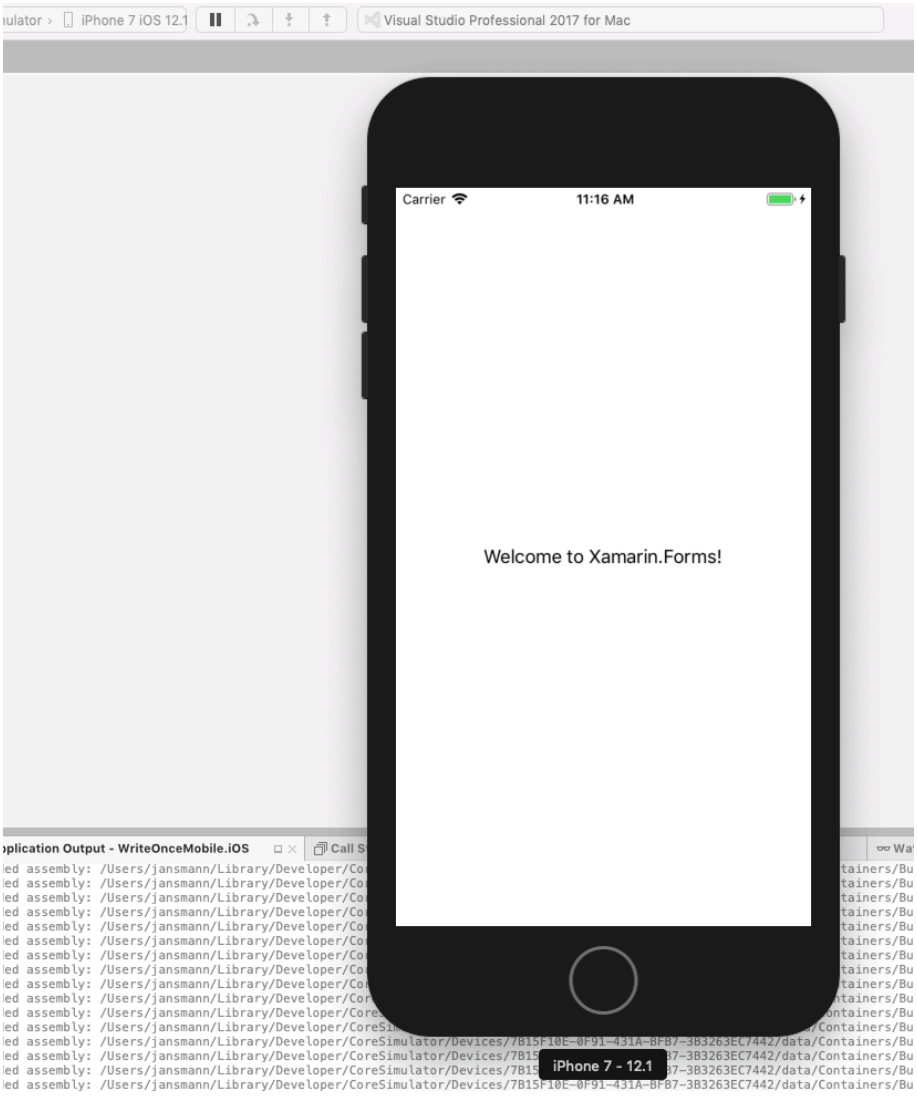


8. Select the particular device you wish to test from the build toolbar:



9. Click the arrow to build and run, or select **Run -> Start With Debugging** from the menu.

10. The iOS Simulator should start and show your app:



Exercise One (A): Launch Android (optional)

Windows – Android

Getting the Android Emulator running on Windows is not a trivial task and varies depending on whether you will be developing solely on a Windows machine, or if you have access to a Mac. Best results are obtained in using the latter option, which is what this example will use.

<u>On the PC:</u>

1. Open your project in Visual Studio on the Windows PC.
2. Ensure the Android project is set at the Startup Project.
3. Switch to the Mac.

<u>On the Mac:</u>

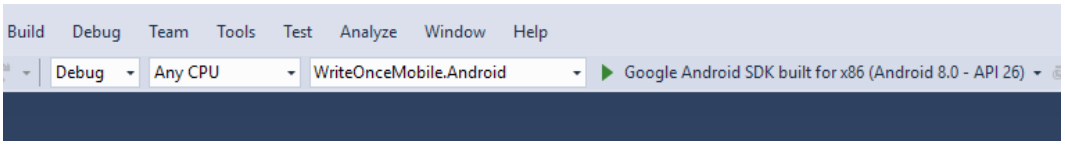
4. Launch Visual Studio for the Mac.
5. From the menu, select **Tools -> Device Manager**.
6. Select and launch the particular version of the emulator you wish to test on.

7. Ensure the image has full booted before continuing to the next step!
8. Launch a console prompt.

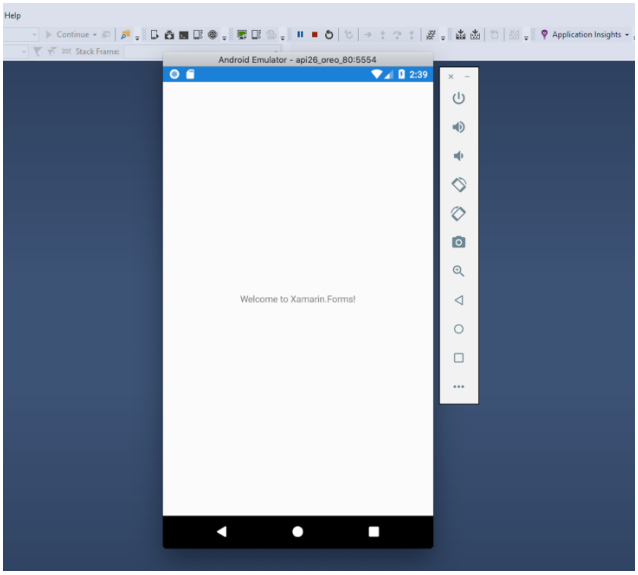
9. Change to the following directory:
   ```
   cd ~/Library/Android/sdk/platform-tools/
   ```
10. Run this command to determine what device is currently running:
    ```
    ./adb devices
    ```
11. You should see a single entry along these lines:
    ```
    emulator-5554 device
    ```
12. If it says “unauthorized” a pop-up may appear asking you to grant permission to debug against this image.
13. Change to the temporary folder:
    ```
    cd /tmp
    ```
14. Create a special pointer that will accept incoming requests to the Mac on a specific port, in this case, 5555, and redirect them to the emulator:
    ```
    rm backpipe
    mkfifo backpipe
    nc -kl 5555 0<backpipe | nc 127.0.0.1 5555 > backpipe
    ```
15. Return to the PC.

<u>Back On the PC:</u>

16. Launch the ADB command prompt from the toolbar in Visual Studio.
17. Run the ADB Connect command to allow the tunnel to be created between your PC and the Mac (note, you will need the exact IP address of the Mac for this command):
    ```
    adb connect <ip address of mac>:5555
    ```
18. The emulator should now appear on the Visual Studio build toolbar:



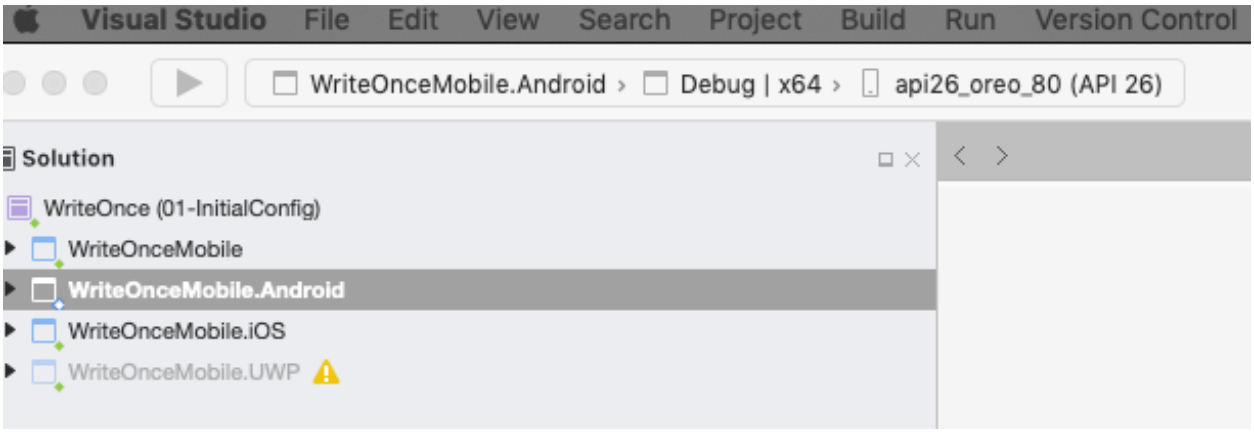
19. Run the project. It will build and launch on the emulator:



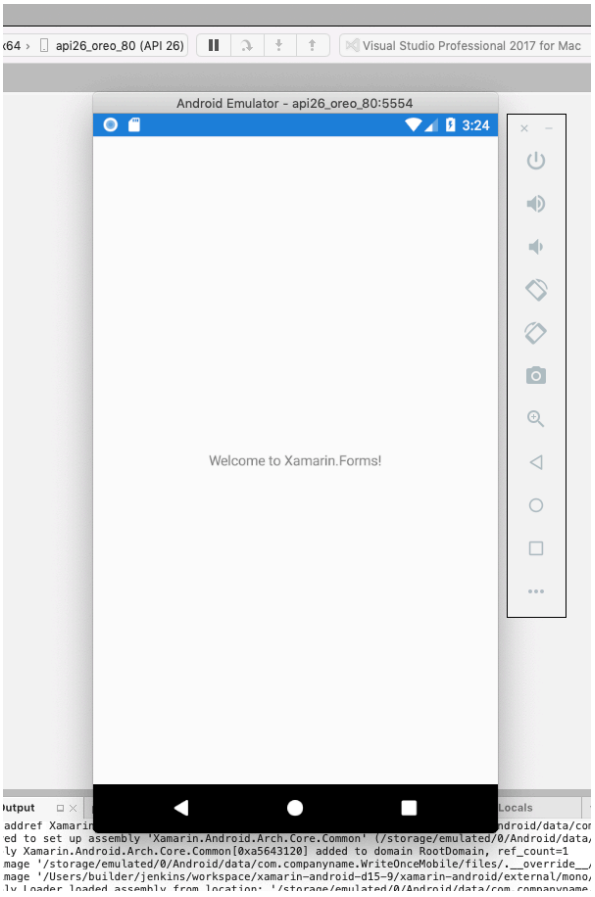
Mac – Android

Building and testing Android apps is far simpler on the Mac version of Visual Studio.

1. Launch Visual Studio for the Mac.
2. Open the project.
3. Right-click the Android project and make it your startup project.
4. Select the appropriate device you wish to test from the selection box at the top of Visual Studio:



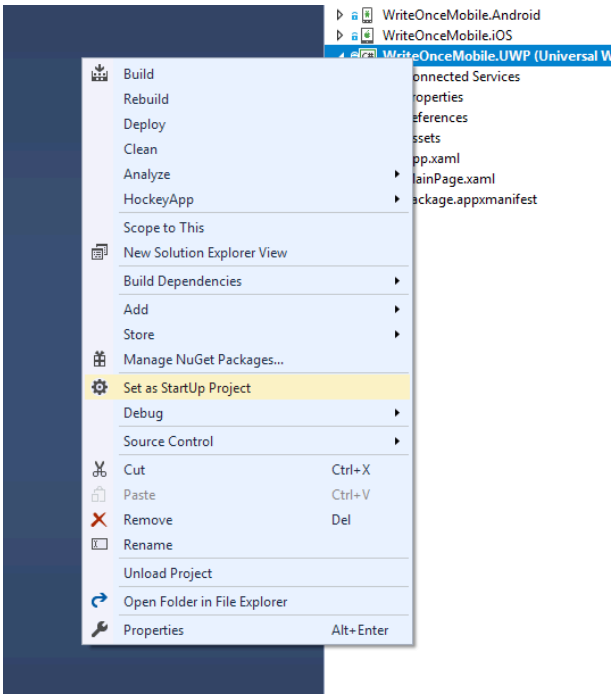
5. Click the run arrow, or select **Run -> Start with Debugging** from the menu.
6. The emulator will launch and your app will start up:



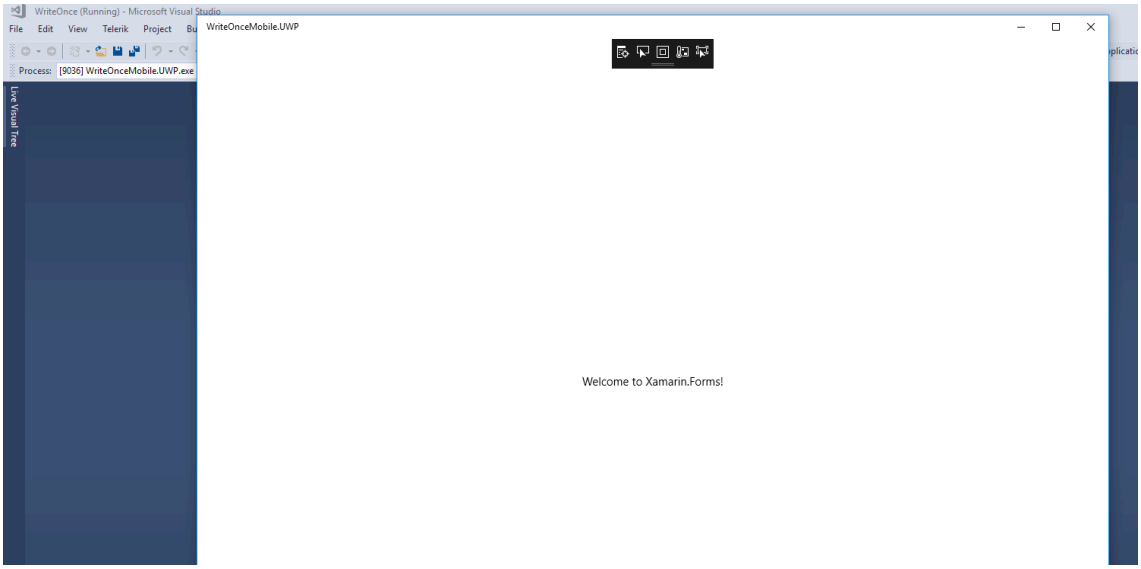
Exercise One (B): Launch Windows (Optional)

If working on a Windows workstation, follow these steps to launch the UWP version of the app for the first time.

1. Open your project in Visual Studio on the Windows PC.
2. Right click on the WriteOnceMobile.UWP project and select **Set as Startup Project**.



3. Launch the project by hitting the green run arrow or selecting **Debug -> Start With Debugging** from the menu.
4. The UWP version of the project will launch locally within the Windows environment:



5. This is actually *not* a mobile emulator; however, code running in this manner will work properly on a Windows mobile device. It is possible to download Windows Phone emulators and "see"

how this program will appear to a mobile device user, but in order to run the emulator, Hyper-V must be installed on the Windows workstation and enabled in BIOS (neither of which are possible on the virtual machine being used for these examples). We'll simply stipulate that "it works" on a full Windows PC. ;-)

Exercise Two: Add a Button (Code First)

Branch: 02-AddButton-Begin

In this exercise, we'll add a simple button and see how it works across the platforms. Although Xamarin allows for drag-and-drop UI development via the XAML interface, all components may also be added via code. The first example will demonstrate this.

1. Open your project in Visual Studio. (Remember, if you're using a Mac, you'll not be able to work on the Windows project.)
2. Expand the tree for the WriteOnceMobile shared project and locate the **App.xaml.cs** file.
3. This is the main class that is called by default when the application starts.
4. Locate the constructor method. It should appear something like this:

```
/// <summary>
/// This is the root page of our application.  When the program launches, items in this section will
/// be run across all three platforms.
/// </summary>
/// <remarks>
/// Some third party components will require you to add material in this method in order to correctly
/// wire them up for use.
/// </remarks>
3 references | Christopher H. Jansmann, 8 days ago | 1 author, 1 change
public App()
{
    //  Base call.
    InitializeComponent();

    //  Load and launch the main page of this application.  We've conveniently
    //  called it "MainPage" but you can specify object of type ContentPage.
    //  If necessary, this call may be overridden and a code-derived view could be passed in.

    //  TODO: 02-Add a button in code here.
    MainPage = new MainPage();
}
```

5. Let's add a button users can click via code. Add the following lines below the **InitializeComponent()** call:

```
//  Let's create a basic stack layout to house some UI Controls.
var layout = new StackLayout
{
    VerticalOptions = LayoutOptions.Center, //   Placement on the vertical plane, scaled for the device.
    Children =
    {
        new Label
        {
            HorizontalTextAlignment = TextAlignment.Center,
            Text = "Welcome to WriteOnce!"
        }
    }
};
```

6. Remove the current lines implementing the **MainPage** object with the following:

```
//  Override the main page layout and add this new one.
MainPage = new ContentPage
{
    Content = layout
};
```

These lines will attach the layout that we created above to the Content property for the page. This removes anything already placed on the Content property, even if added via XAML.

7. Add a button object and set some properties:

```
//
//  Let's create a classy button...
Button button = new Button
{
    Text = "Click Me"
};
```

8. Next, we need to capture the “click” event from the user. We’ll wire it up here, although it could also be done on the UI. Add this code:

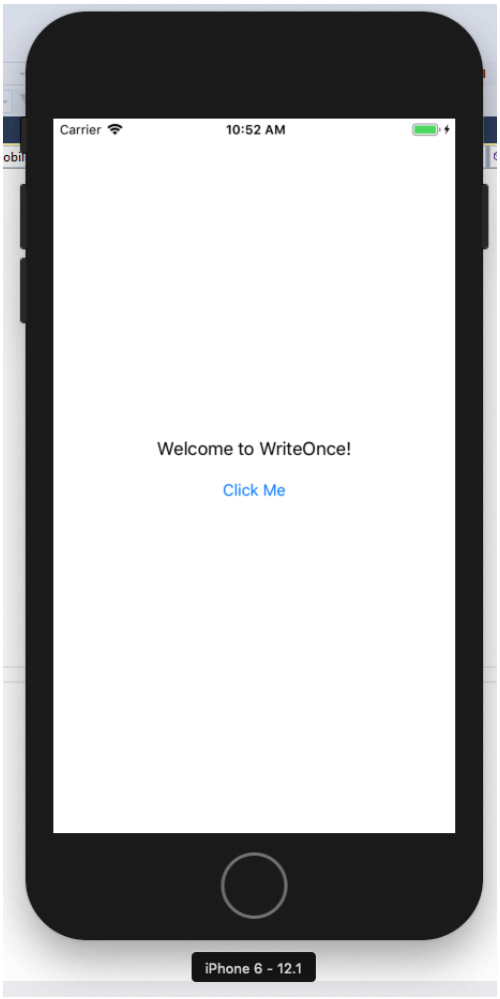
```
//  ...and wire up an event to handle the click...
button.Clicked += async (s, e) => {
    await MainPage.DisplayAlert("Alert", "You clicked me", "OK");
};
```

9. Finally, add this new button to the layout (which has already been added to the Content for the page):

```
//  ...and add this item to the layout.
layout.Children.Add(button);
```

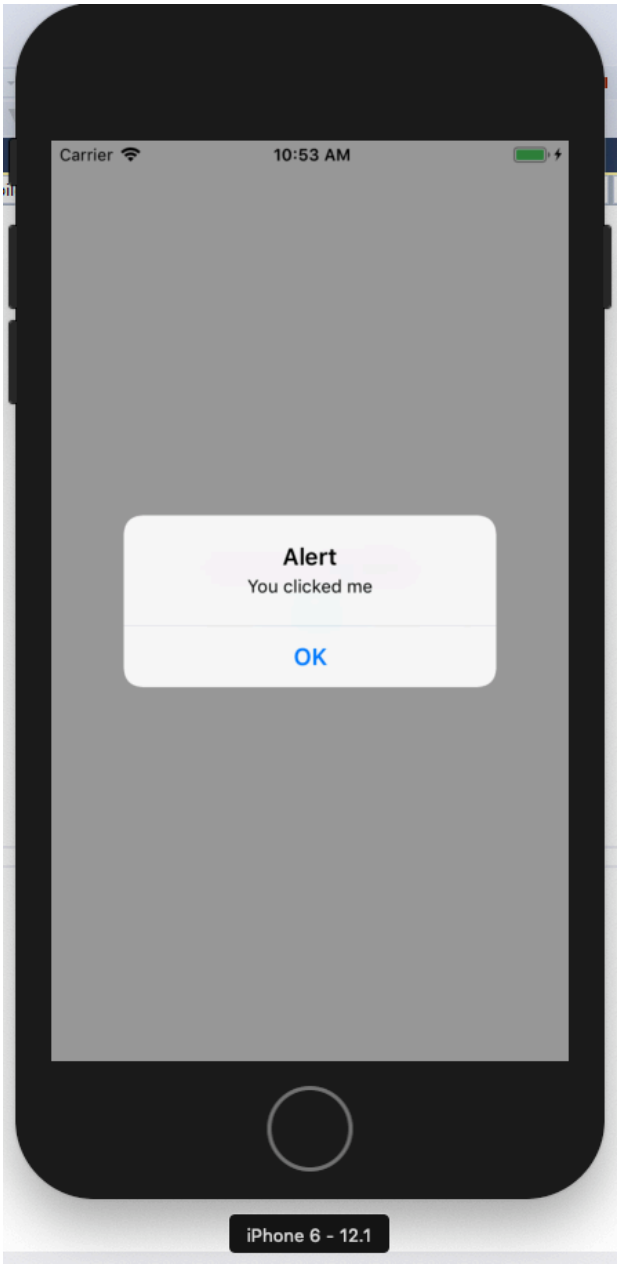
10. Try to build and run the project.

11. First, we'll use iOS to see how the button acts in that environment. You should now see a button on your main screen:



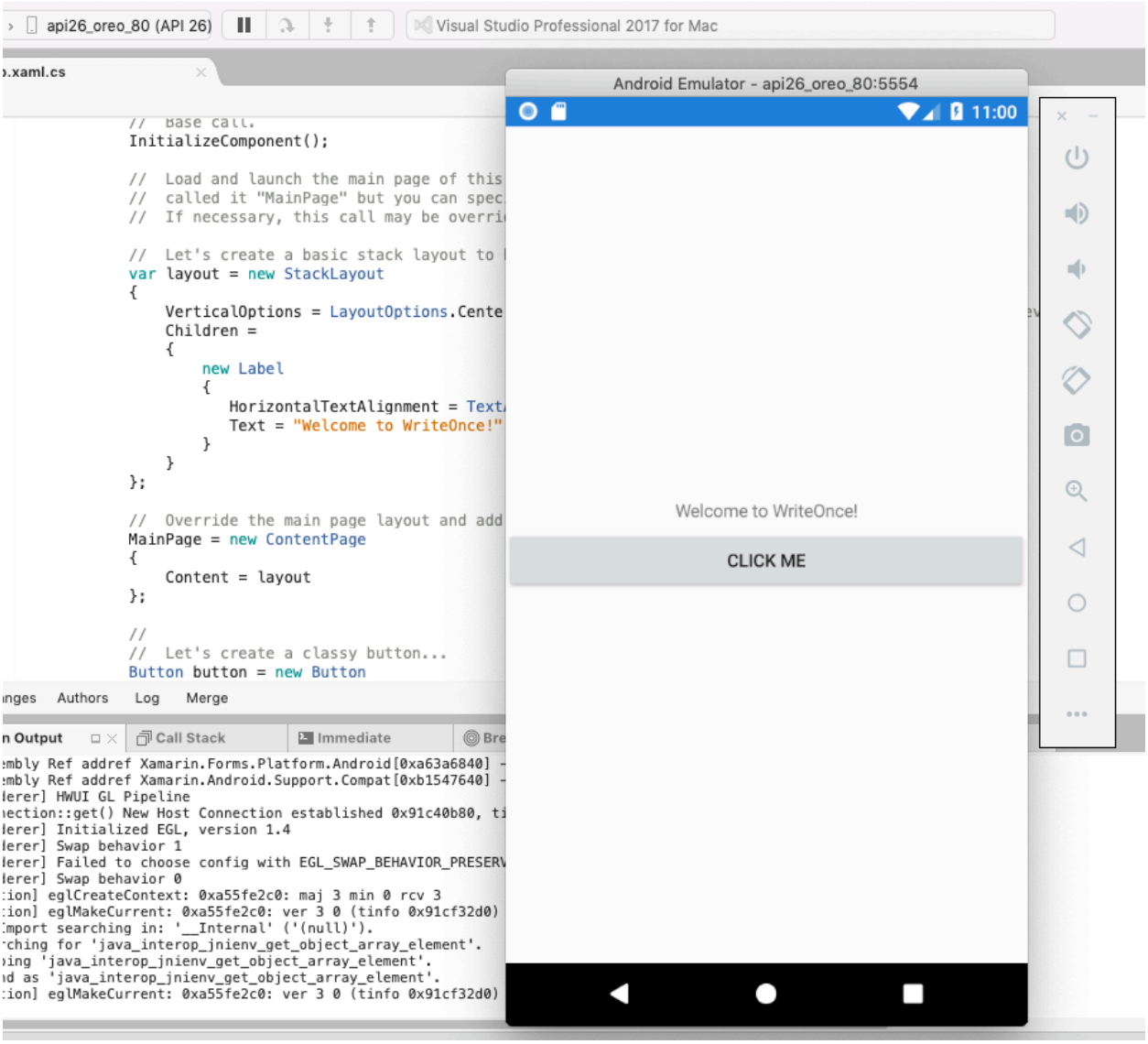
12. Note the button looks and operates as an iOS element.

13. Click the button!



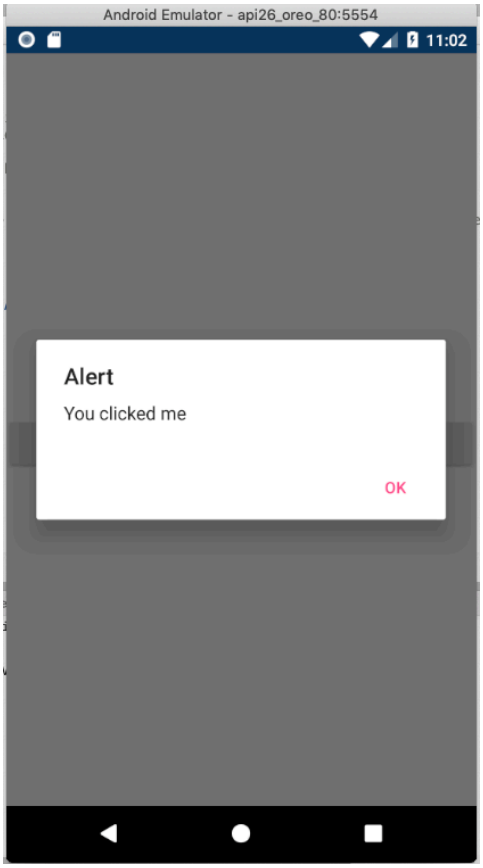
14. This has a very iOS feel to it, and we didn't do anything to force that.

15. Now try Android:



16. Note how with no alterations to the code, the button has taken on the look and feel of a native Android UI element. While we can still make other changes to the button (color, etc.), we did not have to explicitly tell Xamarin to make the button look “droidesque.”

17. Click the button!



18. Again, note that the alert dialog box looks and feels like an Android element – without us having to explicitly code for it.

Exercise Three: Add a Button (Design First)

Branch: 03-AddButtonUI-Begin

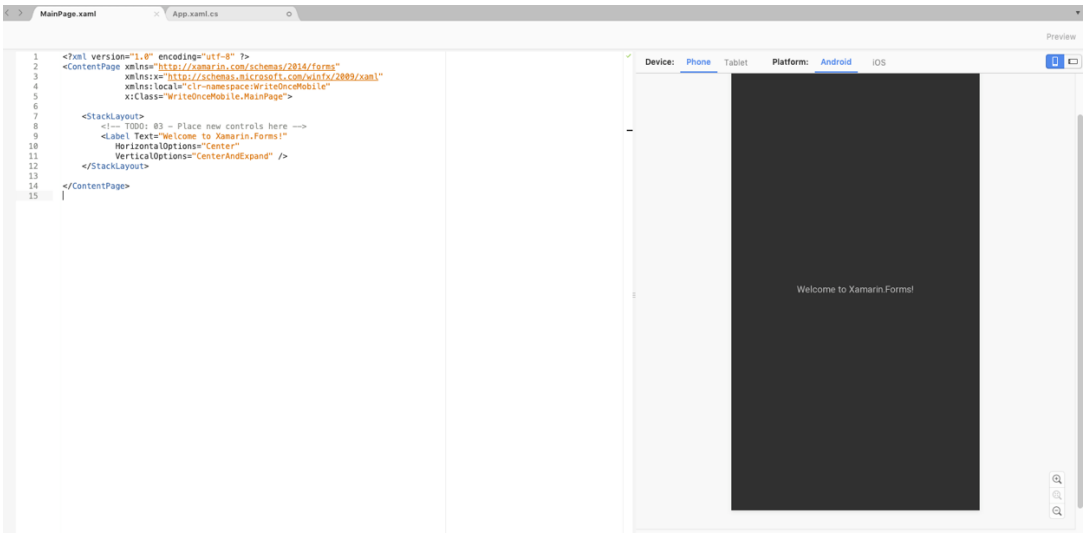
Working with the XAML preview window can simplify layout of forms on mobile devices. While the XAML preview is available on both Windows and Mac platforms, the tools are a little more polished on the Mac side of the house (since that is where Xamarin initially did most of its original development). This example will make use of Visual Studio for the Mac, although it will work very similarly on Visual Studio for the PC. (If you are wondering, XAML stands for Extensible Application Markup Language.)

1. Open your project in Visual Studio for the Mac.
2. If possible, begin with the project as it stood at the end of the last exercise, or get a clean copy of the starting branch (03-AddButonUI-Begin).
3. Within the **WriteOnceMobile** shared project, Open the **App.xaml.cs** file and either comment out or remove the lines we added earlier to create a layout section and button. We'll do this on the UI next.
4. Add back in the original line that creates the MainPage object:

```
/// <summary>
/// This is the root page of our application.  When the program launches, items in this section will
/// be run across all three platforms.
/// </summary>
/// <remarks>
/// Some third party components will require you to add material in this method in order to correctly
/// wire them up for use.
/// </remarks>
public App()
{
    //  Base call.
    InitializeComponent();

    //  Load and launch the main page of this application.  We've conveniently
    //  called it "MainPage" but you can specify object of type ContentPage.
    //  If necessary, this call may be overridden and a code-derived view could be passed in.
    MainPage = new MainPage();

}
```

5. Open the MainPage.xaml file. In Visual Studio for the Mac, by default you'll get a two-pane setup showing the markup (left side) and a live preview (right side) for the device you are targeting. You'll see that in our example, it's still showing the base code we had from creating

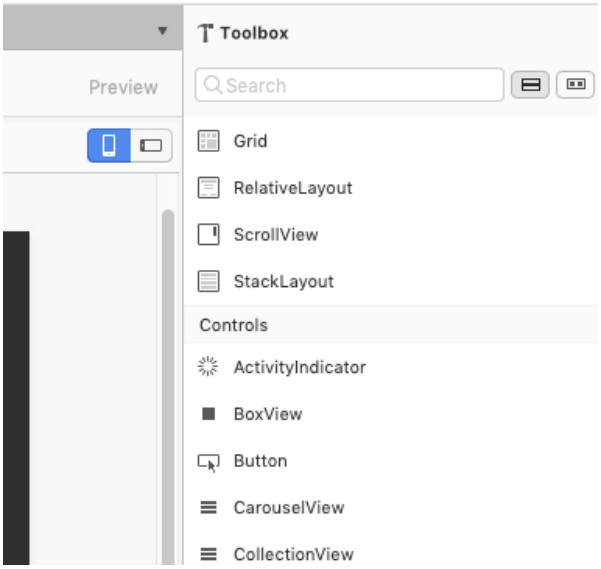
the project.



6. The initial template left us with a StackLayout and a Label. StackLayout is one of multiple layout controls in Xamarin. We'll leave this in place, adjust the label and add back in our button.
7. XAML is a mashup of markup language and XML layout (i.e. eXtensible Markup Language) and has it's own syntax. As of this writing, Xamarin XAML was being collapsed into standard Microsoft XAML notation. The important thing to note here is that every tag must have an opening and a closing tag in order to compile. Intellisense should help, but keep this in mind as you work.
8. You can manually create the button using XAML syntax of

```
<Button
      x:Name="ClickMeButton"
      Text="Click Me (UI Version)">
</Button>
```

or you can drag a new button from the Toolbox and place it on the form



9. Let's also make some adjustments to the StackLayout to make sure it aligns the way we want it to:

```
<StackLayout
    HorizontalOptions="Center"
    VerticalOptions="Center">
```

10. Finally, we need to tell the button that it will be wired into an event. The easiest way to do this is to add another property to the XAML markup. As you start to type in the new key, Intellisense will offer to create the method handler for you. Go ahead and click on it.

```
<Button
    x:Name="ClickMeButton"
    Text="Click Me (UI Version)"
    Clicked=""
>
</Button>

tackLayout>

ntPage>
```

Handle_Clicked

11. VS for the Mac will take you into the "code behind" file for this content page, essentially where all of the logic will reside. This helps with separation of concerns in that in ensures we don't place any logic on the UI itself.

```
namespace WriteOnceMobile
{
    public partial class MainPage : ContentPage
    {
        public MainPage()
        {
            InitializeComponent();
        }
    }
}
```

Add event handler
Use ▲ ▼ to move to another location.
Press ENTER to select the location.
Press ESC to cancel this operation.

12. Arrow to where you want the event to reside and hit enter. A new method stub will appear.

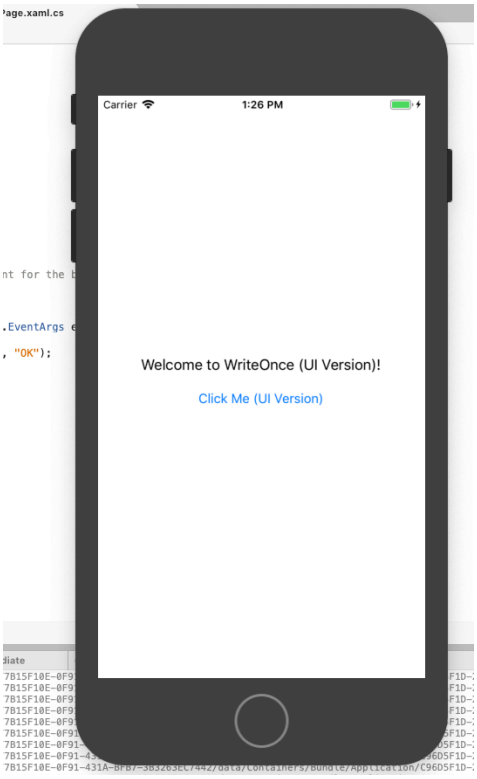
```
void Handle_Clicked(object sender, System.EventArgs e)
{
    throw new NotImplementedException();
}
```

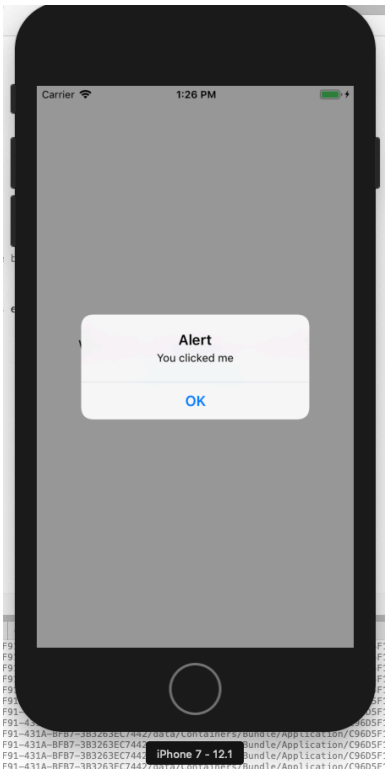
13. Update the code to handle our click event, adding back in our alert message. We do need to also update the method signature as displaying an alert is an asynchronous call:

```
/// <summary>
/// This method will take care of the click event for the button on the page.
/// </summary>
/// <param name="sender">Sender.</param>
/// <param name="e">E.</param>
async void Handle_Clicked(object sender, System.EventArgs e)
{
    await DisplayAlert("Alert", "You clicked me", "OK");
}
```

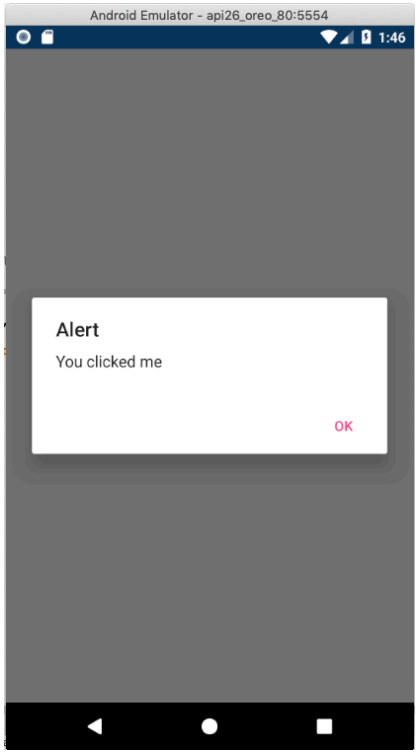
14. Save everything!

15. Test it out. We'll start with iOS and do the usual **Debug -> Start with Debugging** option from the menu.





Looks good on iOS! How about Android?

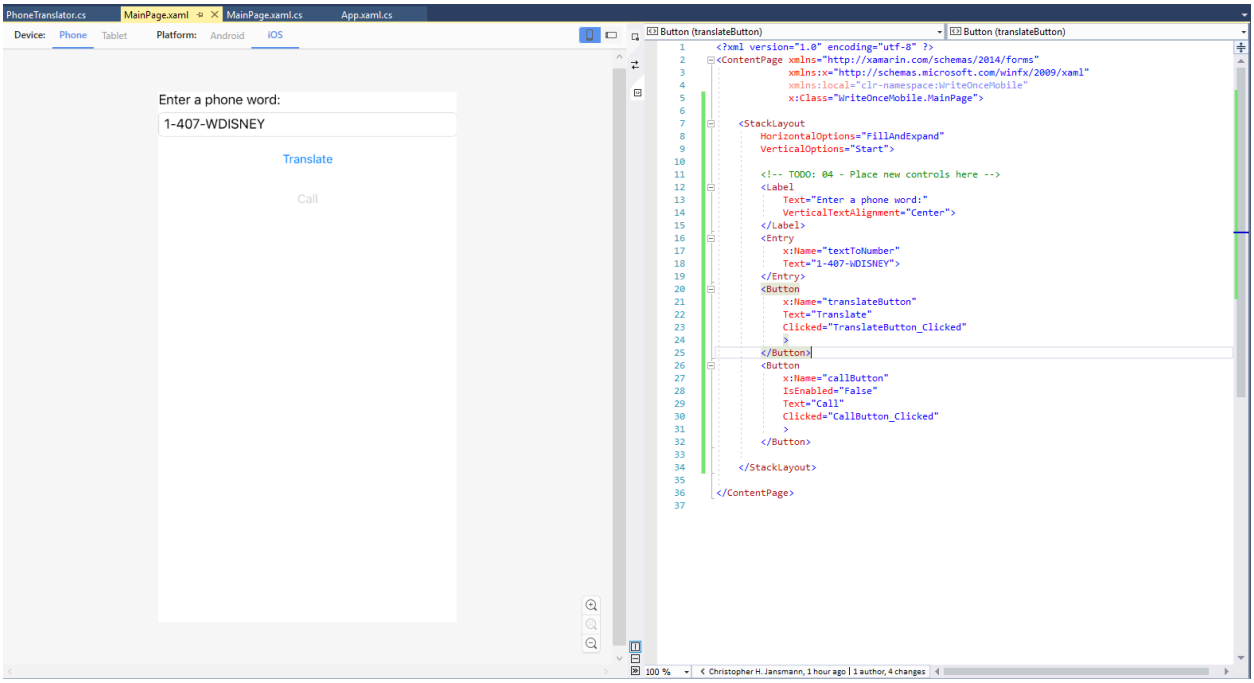


Exercise 4: Interaction

Branch: 04-PhoneWord-Begin

This exercise will add some interactivity to the application, asking for input from the user and then making use of the data. In this case, we will translate an alphanumeric phone number into an actual number that can be dialed by the device (if dialing is allowed).

1. Launch Visual Studio and open your project.
2. Get a clean copy of the branch, 04-PhoneWord-Begin, as this contains some additional resources that will be helpful for completion of the task.
3. In the **WriteOnceMobile** shared project, open the **MainPage.xaml** file. We want to add the following:
    a. Label with instructions
    b. Entry box to accept input
    c. Two button elements
4. Here's how the XAML should look:



5. Note how the XAML live preview updates as you make changes in the XAML code.

6. On code behind for this file, adjust the constructor for MainPage to including a setting for Padding that will adjust based on the platform:

```
//  Minor UI Tweak - this will ensure that we have appropriate padding around the
//  UI across all devices.
PaddingAmount = 0.0;
switch(Device.RuntimePlatform)
{
    case Device.iOS:
        PaddingAmount = 40;
        break;
    default:
        PaddingAmount = 20;
        break;
}

//  Set it.
this.Padding = PaddingAmount;
```

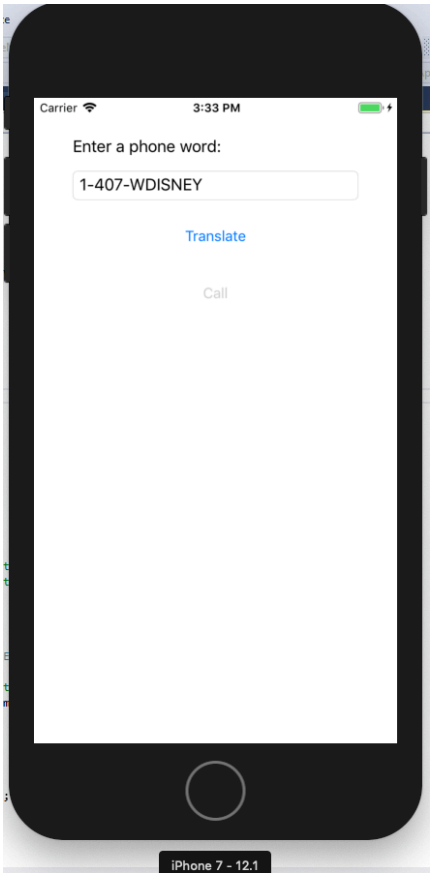
7. Add a class-level variable to store the translated value:

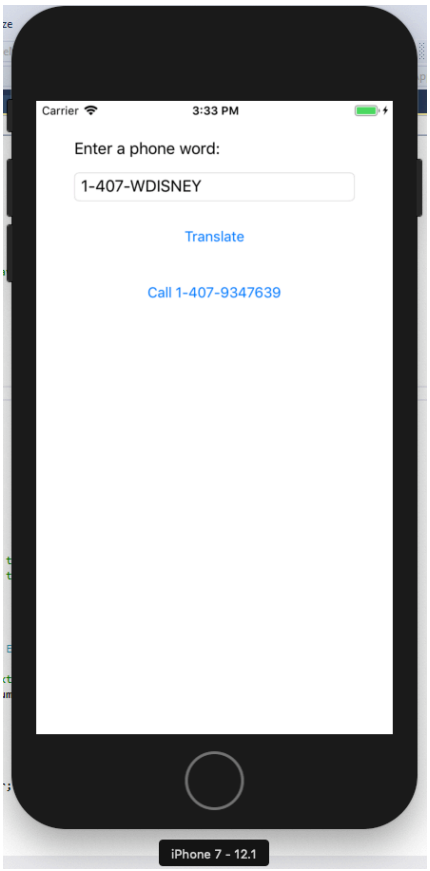
```
//  Placeholder for the translation
3 references | 0 changes | 0 authors, 0 changes
public string TranslatedNumber { get; set; }
```

8. And here is what your events should look like:

```
/// <summary>
/// Examines the text that was entered and attempts to translate it to a
/// usable phone number.  If it can be used, enable the calling button.
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
0 references | 0 changes | 0 authors, 0 changes
private void TranslateButton_Clicked(object sender, EventArgs e)
{
    //  Use our helper function to translate the text to a number
    TranslatedNumber = Core.PhonewordTranslator.ToNumber(textToNumber.Text);

    //  A little bit of error handling...
    if (!string.IsNullOrEmpty(TranslatedNumber))
    {
        callButton.IsEnabled = true;
        callButton.Text = "Call " + TranslatedNumber;
    }
    else
    {
        callButton.IsEnabled = false;
        callButton.Text = "Call";
    }
}
```

9. Run the project and try it out:





10. Hitting the **Call** button should trigger an error, but we will correct that on the next exercise.
11. If you wish, change to the Android project, rebuild and see the differences in how the interface is presented.

Exercise 5: Call the Number

Branch: 05-CallPhone-Begin

We'll wrap up the final part of this project by setting up the code that allows us to dial the number that has been translated. This will also allow us to see how some minor platform specific adjustments can be made in each of the specific device projects.

1. Begin in Visual Studio, and ensure you have a clean copy of the branch for this exercise (05-CallPhone-Begin).
2. Inside the **WriteOnceMobile** shared project, create a new class that will house the interface that will be implemented in each of our mobile projects. Name the file *IDialer.cs*.
3. Make sure the file exists in the top-level namespace of **WriteOnceMobile**.

```
namespace WriteOnceMobile
{
    /// <summary>
    /// Interface that will be implemented in a platform-specific manner
    /// for each device.
    /// </summary>
    0 references | 0 changes | 0 authors, 0 changes
    interface IDialer
    {
        //  TODO: 05 - Add methods to implement.
    }
}
```

4. Add a single method to the interface declaration that takes a string input and returns a Boolean:

```
//  Method that will be implemented on each platform to trigger the dial.
0 references | 0 changes | 0 authors, 0 changes
bool Dial(string number);
```

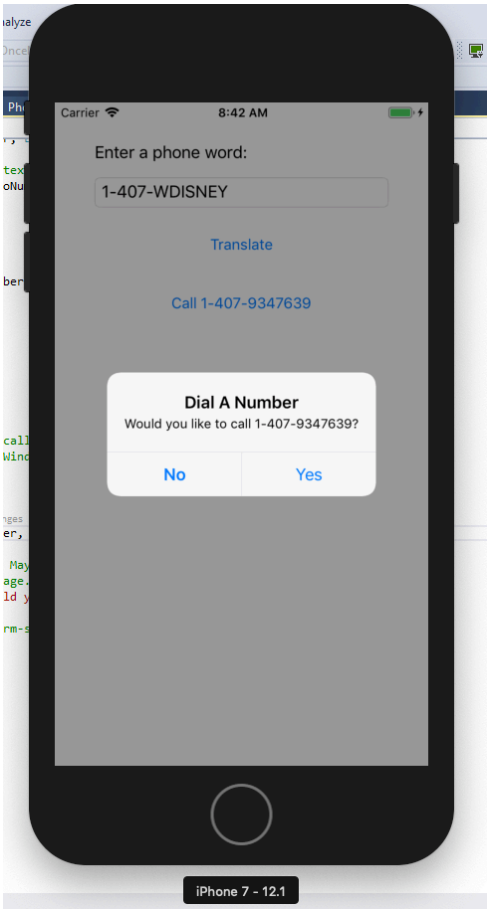
5. Return to the **MainPage.xaml.cs** code behind file and locate the method that we wired up to *CallButton* – we need to make a modification to implement the dialing code, first by marking the method as asynchronous:

```
private async void CallButton_Clicked(object sender, EventArgs e)
```

6. Next, add in a stub of code to trigger a verification dialog when dialing:

```
//  TODO: 05 - Add implementation code here.  May need some platform specific help (hint, hint)
//  First, let's try to trigger a dialog message.  If they respond correctly, dial.
if (await DisplayAlert("Dial A Number", $"Would you like to call {TranslatedNumber}?","Yes","No"))
{
    //  TODO: Dial!  (This will be the platform-specific implementation call)
}
```

7. Build and test what we have so far, using iOS or Android. You should get:



8. However, at this point, we are still not actually dialing on the device. This is where the platform specific code comes in; while Xamarin.Forms is capable of abstracting away much of the underlying device mechanics, there are few times when you will need to make very specific calls that are unique to each platform. Dialing, not surprisingly, is one of them.
9. Start with the **WriteOnceMobile.iOS** project and locate the **PhoneDialer.iOS.cs** class.

10. The class is straightforward: it implements the *iDialer* interface we stubbed in on the shared class, and then implements the **Dial** method using iOS-specific calls.

```
/// <summary>
/// This helper class includes goodies that will perform platform-specific
/// operations on iOS.  Interface needs to be defined in the shared class.
/// </summary>
0 references | Christopher H. Jansmann, 56 minutes ago | 1 author, 1 change
public class PhoneDialer : IDialer
{
    /// <summary>
    /// Dial the phone, iOS Style!
    /// </summary>
    /// <param name="number"></param>
    /// <returns></returns>
    0 references | Christopher H. Jansmann, 56 minutes ago | 1 author, 1 change
    public bool Dial(string number)
    {
        //  Simplier call than Android; iOS treats it as a URL type of call.
        return UIApplication.SharedApplication.OpenUrl(new NSUrl("tel:" + number));
    }
}
```

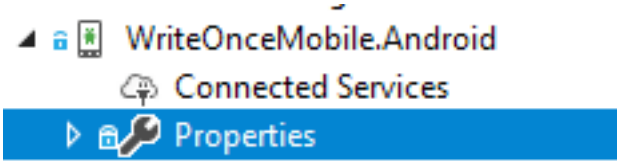
11. Next, locate the **PhoneDialer.Droid.cs** class in the **WriteOnceMobile.Android** project. Note that it is longer and has a secondary helper method.
12. First, the **Dial** method is implemented, within which the Droid app must first obtain a reference to the running context (or view). This is a different threading mechanism than iOS, but Xamarin.Forms is actually keeping track of it for us under the hood. We simply need to grab the current context:

```
/// <summary>
/// Dial the phone, Android Style!
/// </summary>
0 references | Christopher H. Jansmann, 58 minutes ago | 1 author, 1 change
public bool Dial(string number)
{
    //  Shift to the correct Android application context (post Xamarin.Forms 2.5)
    var context = Android.App.Application.Context;
    if (context == null)
        return false;

    var intent = new Intent(Intent.ActionCall);
    intent.SetData(Uri.Parse("tel:" + number));

    if (IsIntentAvailable(context, intent)) {
        context.StartActivity(intent);
        return true;
    }

    return false;
}
```

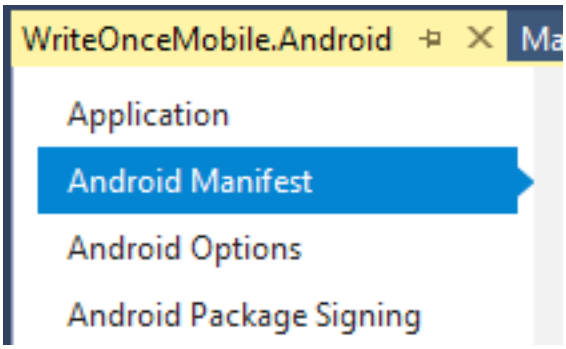
13. However, the helper method is needed to ensure we are not blocked in making our call:

```
/// <summary>
/// Checks if an intent can be handled.  If not, the call will
/// return false and prevent the operation from taking place.
/// </summary>
1 reference | Christopher H. Jansmann, 58 minutes ago | 1 author, 1 change
public static bool IsIntentAvailable(Context context, Intent intent)
{
    var packageManager = context.PackageManager;

    var list = packageManager.QueryIntentServices(intent, 0)
        .Union(packageManager.QueryIntentActivities(intent, 0));
    if (list.Any())
        return true;

    TelephonyManager mgr = TelephonyManager.FromContext(context);
    return mgr.PhoneType != PhoneType.None;
}
```

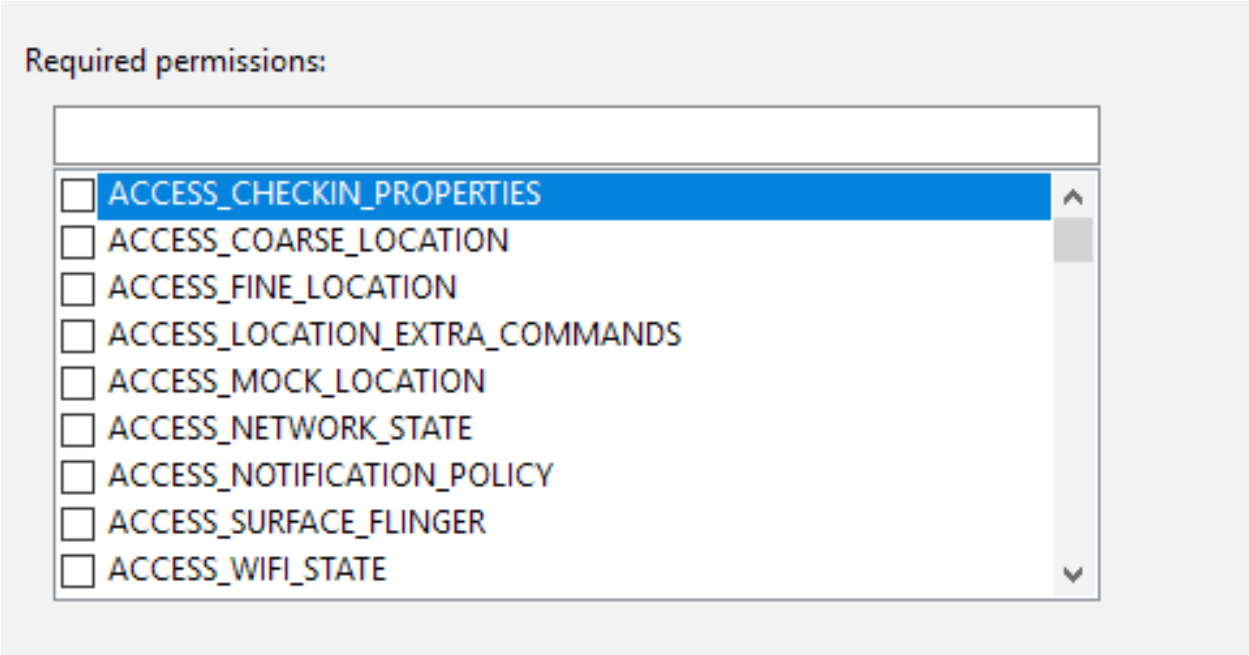
14. In this sense, the intent is the operation we wish to perform, i.e. dialing the number.
15. Before we leave the Android application, there is one last change we need to make. Droid apps require permission to access the phone functions, so we need to add this request to the manifest file for the application. In Visual Studio for Windows, double click on the *Properties* wrench to open the property window for the app:



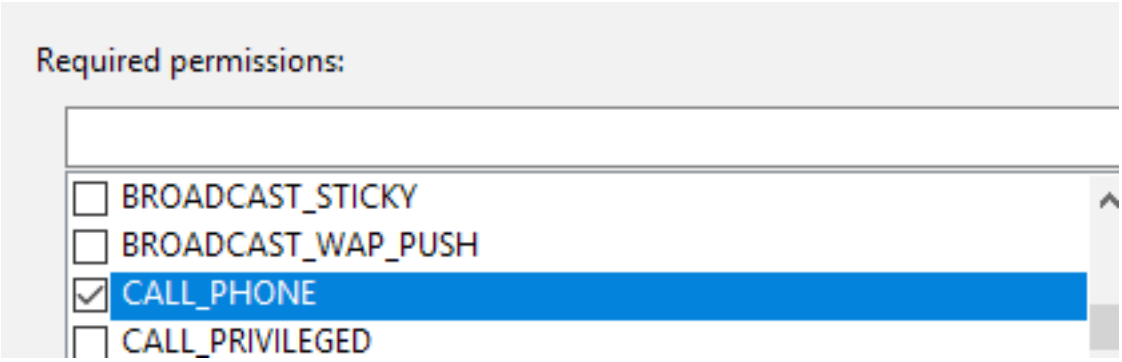
16. Select the *Android Manifest* from the tree on the right:



17. Scroll down to the *Required Permissions* selection window:



18. Further scroll through the list and find the *CALL_PHONE* privilege and select it:

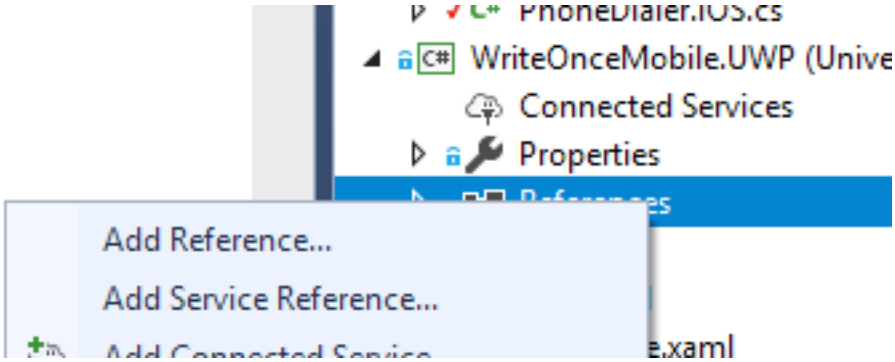


19. Save the file and close the property window.
20. Finally, on the **WriteOnceMobile.UWP** application, locate the **PhoneDialer.WinPhone.cs** class.
21. You'll note that the file is replete with red squiggly lines indicating a compile error.

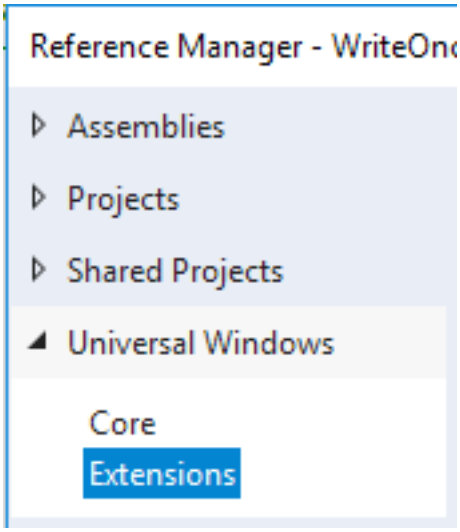
```
PhoneCallTask phoneCallTask = new PhoneCallTask
```

22. By default, UWP projects do not specify the correct Windows Mobile SDK when Xamarin creates the project; we will first need to add a reference before implementing our platform-specific dialer.

23. Right click on the **References** node in the project and select **Add Reference**.

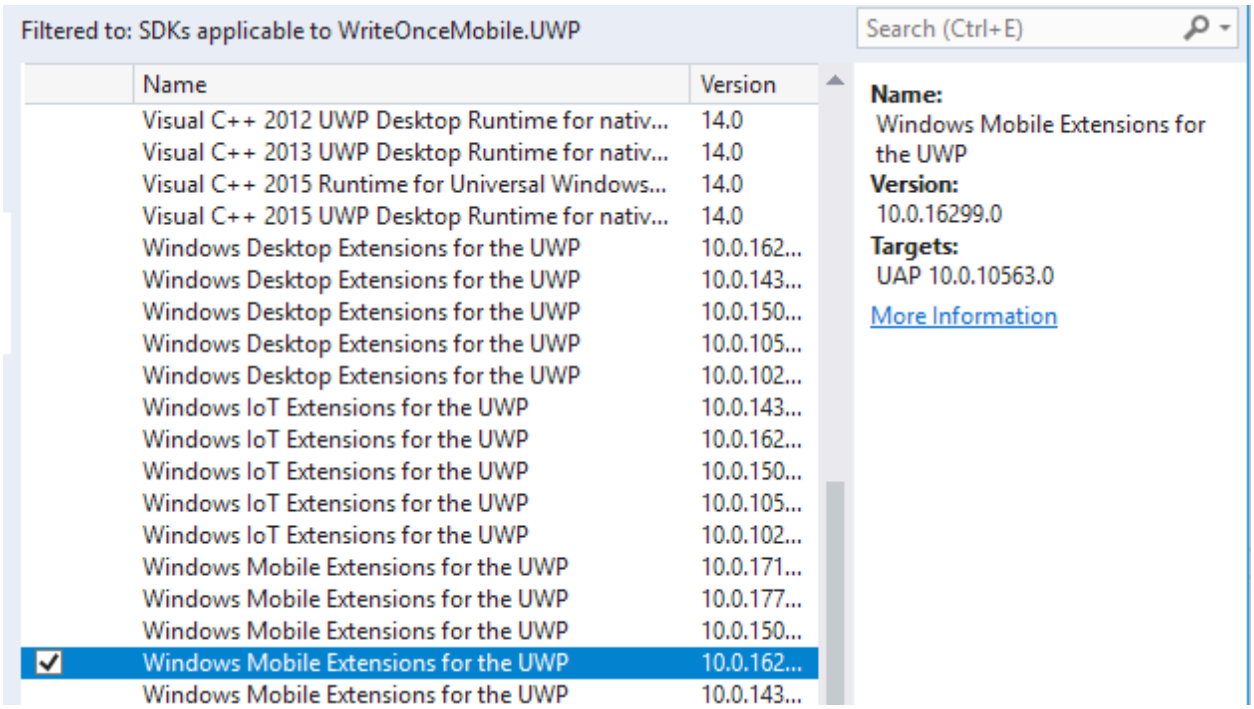


24. Switch to the **Universal Windows -> Extensions** node on the property window.



25. Scroll the list to find the entry for **Windows Mobile Extensions for UWP**. There will be multiple, ordered by the SDK versions loaded on your machine. Select the one that will be most compatible with the devices you are planning on running this app on (in this case, we are going

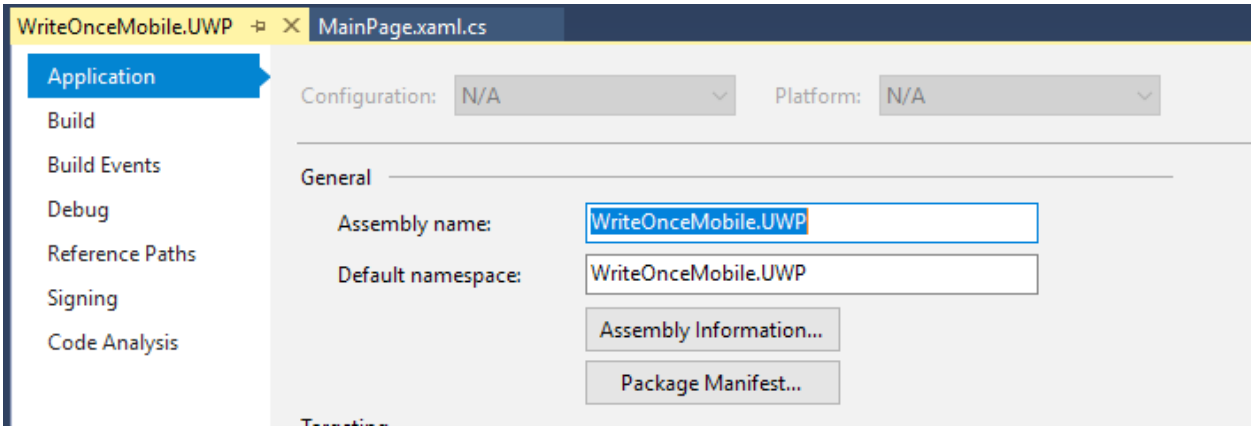
with 10.0.16299):



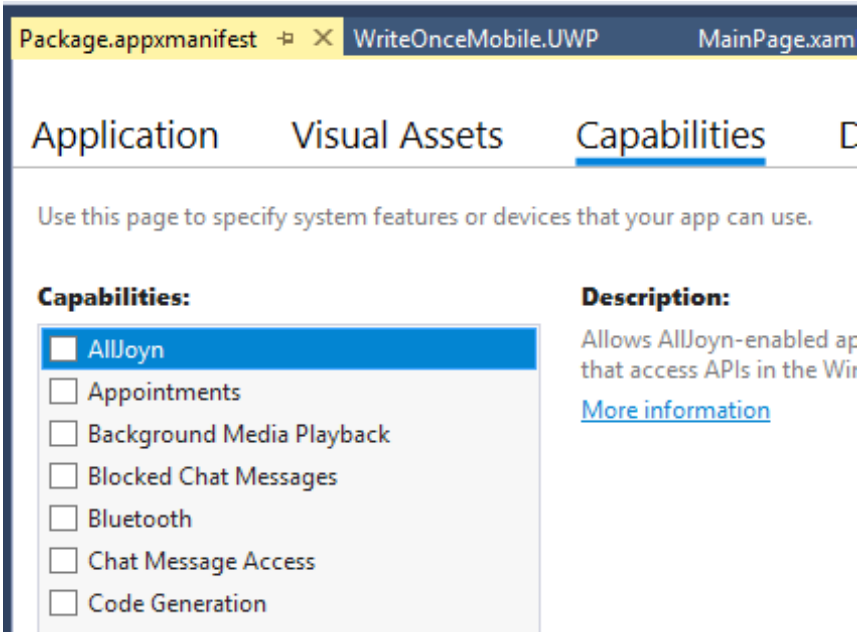
26. Click OK and return to our PhoneDialer.UWP.cs class.
27. Implement the dialer with a single line:

```
/// <summary>
/// Dial, Windows UWP style!
/// </summary>
/// <param name="number"></param>
/// <returns></returns>
0 references | Christopher H. Jansmann, 1 hour ago | 1 author, 1 change
public bool Dial(string number)
{
    //  Windows must reference the Windows Mobile Extensions for UWP in order to make the call.
    //  Once the correct SDK is referenced, this is the correct line for UWP apps to trigger the dialing.
    Windows.ApplicationModel.Calls.PhoneCallManager.ShowPhoneCallUI(number, "Call");

    return true;
}
```

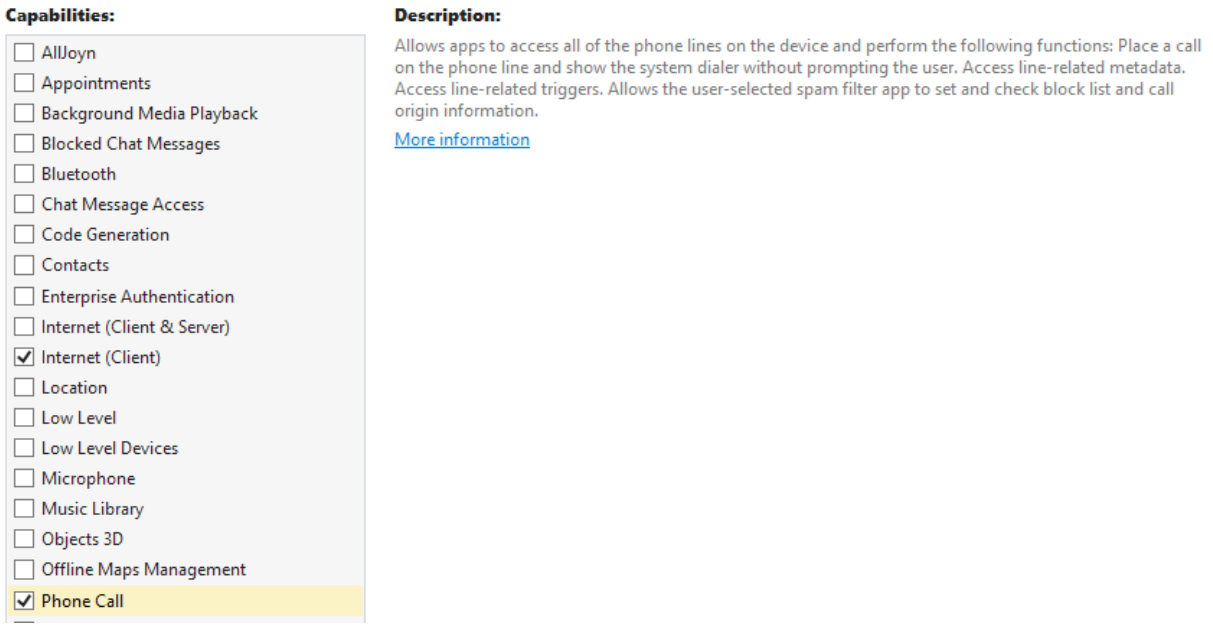
28. Finally, we also need to allow the Windows app access to the phone, specifically the ability to dial. Double click on the **Properties** wrench, the click the **Package Manifest** button.



29. Click to the **Capabilities** tab.



30. Scroll the box to get to the **Phone Call** option, and select it.



31. Save the file and close it out.
32. Rebuild the project, and ensure everything compiles.
33. Return to **MainPage.xaml.cs** in our shared project. Time to implement the dialer!
34. Part of the Xamarin.Forms magic allows us to use Dependency Injection to retrieve the correct interface for the platform that is running out application. This magic can only occur, however, if each of our implemented interfaces has the following assembly attribute just above the namespace declaration:

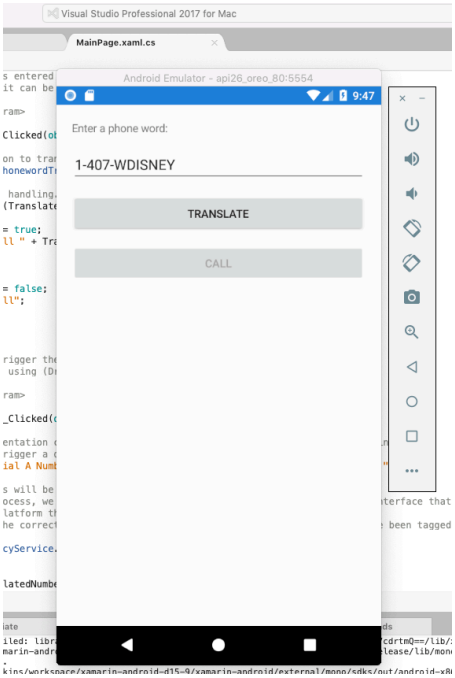
```
// This same attribute definition needs to go in each platform-specific file.
[assembly: Dependency(typeof(PhoneDialer))]
namespace WriteOnceMobile.Droid
```

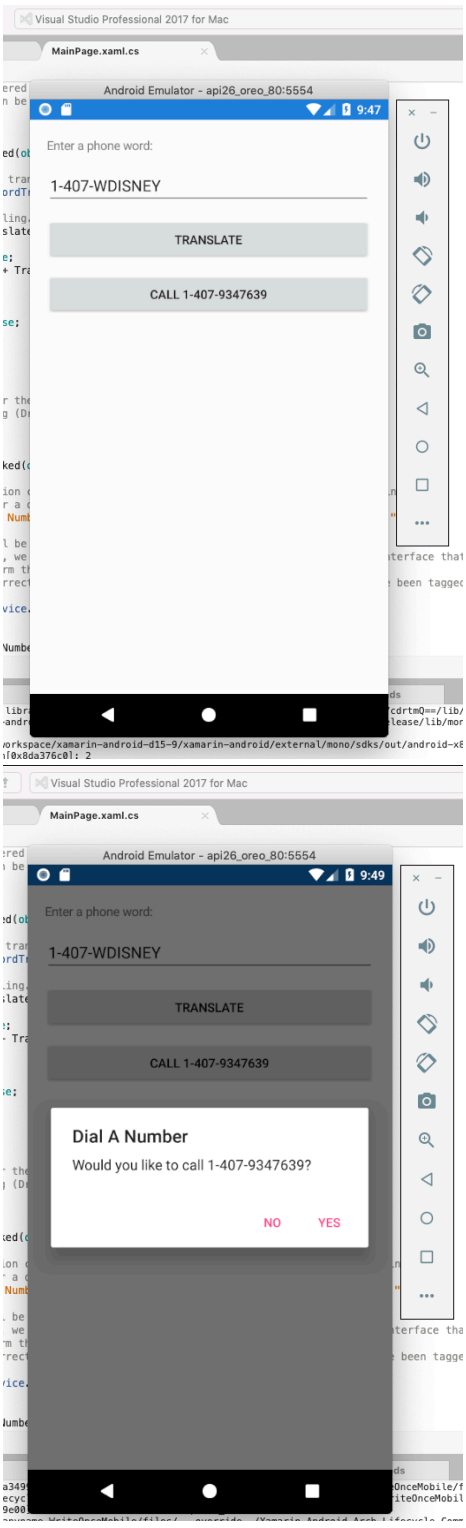
35. This line is the same across all of the apps (iOS, Android, UWP). Make sure this line exists in PhoneDialer.iOS.cs, PhoneDialer.Droid.cs and PhoneDialer.UWP.cs. You may be prompted to add a using statement for Xamarin.Forms.

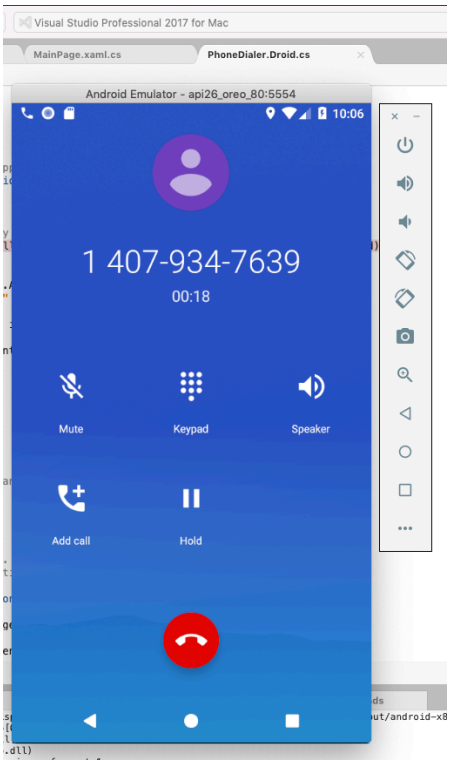
36. Next, add this code to the **CallButton_Clicked** method in MainPage.xaml.cs:

```
/// <summary>
/// Event handler that will trigger the correct "call" function for the
/// platform that the user is using (Droid, iOS, Windows).
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
0 references | Christopher H. Jansmann, 1 hour ago | 1 author, 2 changes
private async void CallButton_Clicked(object sender, EventArgs e)
{
    //  TODO: 05 - Add implementation code here.  May need some platform specific help (hint, hint)
    //  First, let's try to trigger a dialog message.  If they respond correctly, dial.
    if (await DisplayAlert("Dial A Number", $"Would you like to call {TranslatedNumber}?","Yes","No"))
    {
        //  TODO: Dial!  (This will be the platform-specific implementation call)
        //  To aid in this process, we are going to use Dependency injection to implement the interface that is
        //  specific to the platform that is running this code.
        //  DI will located the correct interface for us magically, but only if the classes have been tagged
        //  correctly.
        var dialer = DependencyService.Get<IDialer>();
        if (dialer != null)
        {
            dialer.Dial(TranslatedNumber);
        }
    }


}
```

37. In order to actually see the dialing simulation, it's necessary to use the Android emulator – iOS and UWP will simply take the request and ignore it. Make the **WriteOneMobile.Android** project your startup code and launch the simulator.
38. You should see:




Visual Studio Professional 2017 for Mac
MainPage.xaml.cs
Android Emulator - api26_oreo_80:5554
9:47
Enter a phone word:
1-407-WDISNEY
TRANSLATE
CALL 1-407-9347639
Visual Studio Professional 2017 for Mac
MainPage.xaml.cs
Android Emulator - api26_oreo_80:5554
9:49
Enter a phone word:
1-407-WDISNEY
TRANSLATE
CALL 1-407-9347639
Dial A Number
Would you like to call 1-407-9347639?
NO
YES



39. Yay!
40. Note that simulators are just that, simulators, and don't replace the need for actually testing on real devices. There are multiple online toolsets that can aid in this.