Hello, Android

Introduction to Android Development with Xamarin

In this two-part guide, we build our first Xamarin.Android application using Xamarin Studio or Visual Studio and develop an understanding of the fundamentals of Android application development with Xamarin. Along the way we'll introduce the tools, concepts, and steps required to build and deploy a Xamarin.Android application.

Hello, Android Deep Dive

In the Hello, Android Quickstart, we built and ran our first Xamarin.Android application. Now it's time to develop a deeper understanding of how Android applications work so we can build more sophisticated programs. In this guide we review the steps that we took in the Hello, Android walkthrough so that we can understand what we did, and begin to develop a fundamental understanding of Android application development.

We will touch upon the following topics:

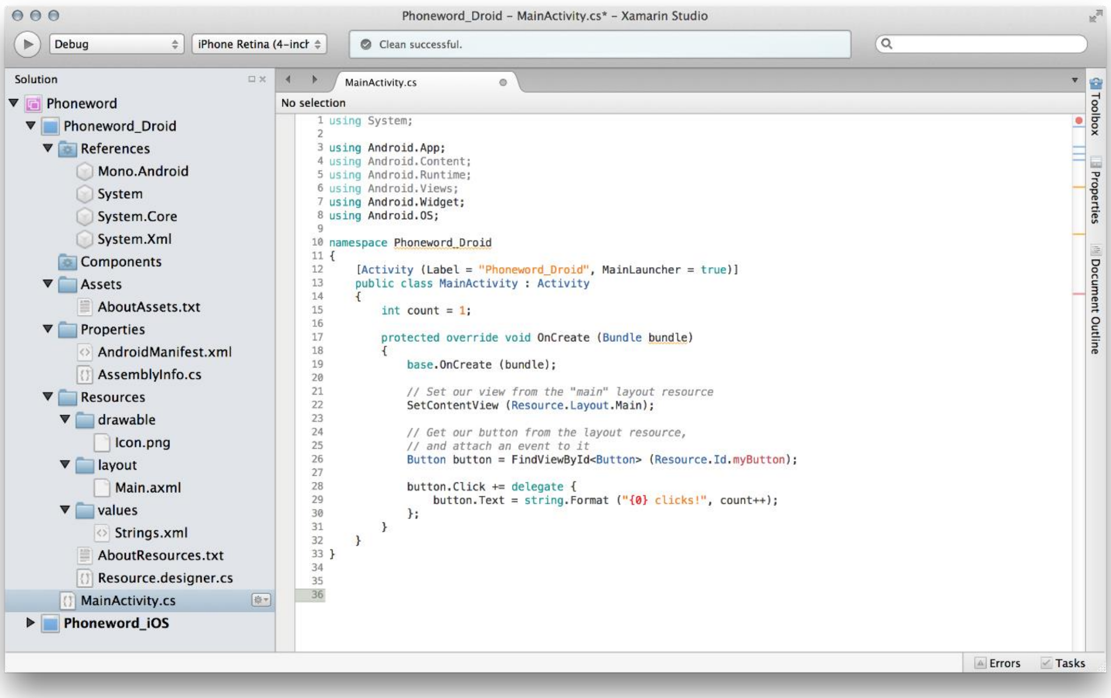
- **Introduction to Xamarin Studio** – Introduction to Xamarin Studio and creating a new application.
- **Introduction to Visual Studio** – Introduction to Visual Studio and creating a new application.
- **Anatomy of a Xamarin.Android Application** - Tour of the essential parts of a Xamarin.Android application.
- **App Fundamentals and Architecture Basics** – Introduction to Activities, the Android Manifest, and the general flavor of Android development.
- **User Interface (UI)** – Creating user interfaces with the Android Designer.
- **Activities and the Activity Lifecycle** – An introduction to the Activity Lifecycle and wiring up the user interface in code.
- **Testing, Deployment, and Finishing Touches** – Complete your application with advice on testing, deployment, generating artwork, and more.

This guide helps you develop the skills and knowledge required to build a single-screen Android application. After working through it, you should have an understanding of the different parts of a Xamarin.Android application and how they fit together.

Introduction to Xamarin Studio

Xamarin Studio is a free, open-source IDE similar to Visual Studio. It features a fully integrated visual designer, a text editor complete with refactoring tools, an assembly browser, source code integration, and more. In this guide we'll learn to use some basic Xamarin Studio features, but if you're new to Xamarin Studio you will want to check out a more in-depth Introduction to Xamarin Studio.

Xamarin Studio follows the Visual Studio practice of organizing code into *Solutions* and *Projects*. A Solution is a container that can hold one or more Projects. A Project can be an application (such as iOS or Android), a supporting library, a test application, and more. In our Phoneword app, we added a new Android Project using the *Android Application* template to the Phoneword Solution we created in the Hello, Android guide. Our Solution looked like this:

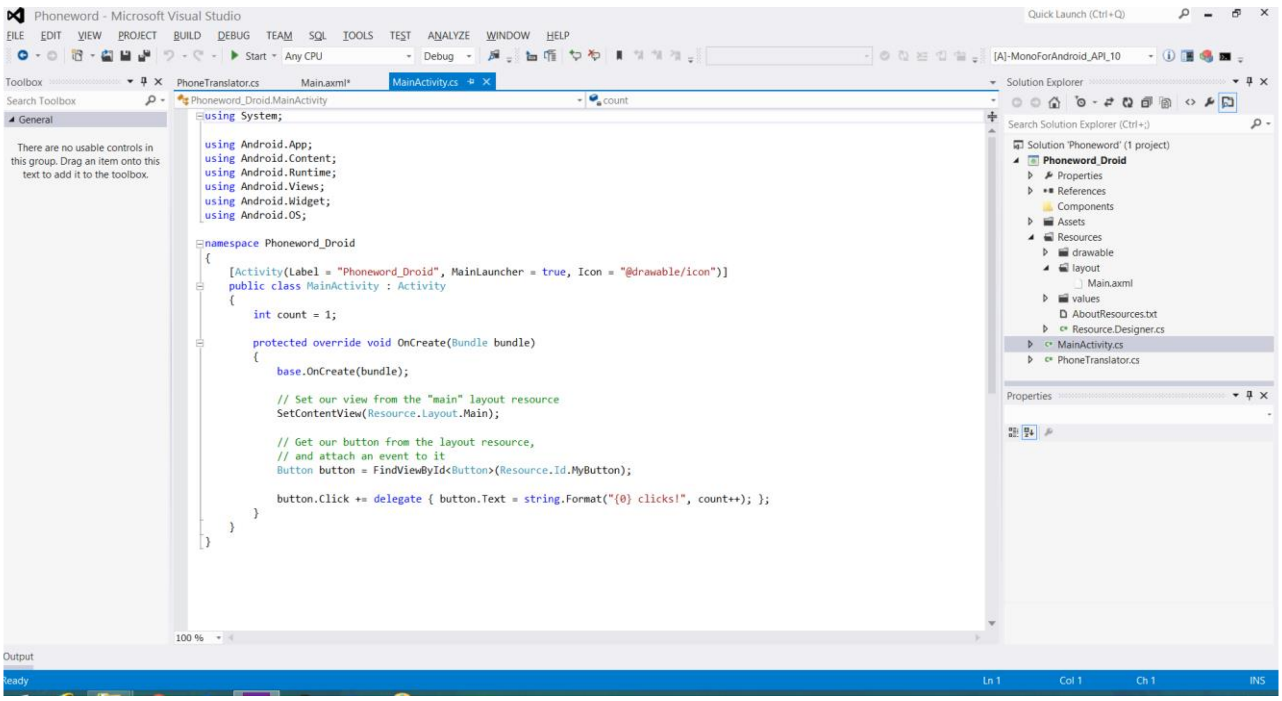


```
using System;

using Android.App;
using Android.Content;
using Android.Runtime;
using Android.Views;
using Android.Widget;
using Android.OS;

namespace Phoneword_Droid
{
    [Activity (Label = "Phoneword_Droid", MainLauncher = true)]
    public class MainActivity : Activity
    {
        int count = 1;

        protected override void OnCreate (Bundle bundle)
        {
            base.OnCreate (bundle);

            // Set our view from the "main" layout resource
            SetContentView (Resource.Layout.Main);

            // Get our button from the layout resource,
            // and attach an event to it
            Button button = FindViewById<Button> (Resource.Id.myButton);

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

Introduction to Visual Studio

Visual Studio is a powerful IDE from Microsoft. It features a fully integrated visual designer, a text editor complete with refactoring tools, an assembly browser, source code integration, and more. In this guide we'll learn to use some basic Visual Studio features with the Xamarin plug-in.

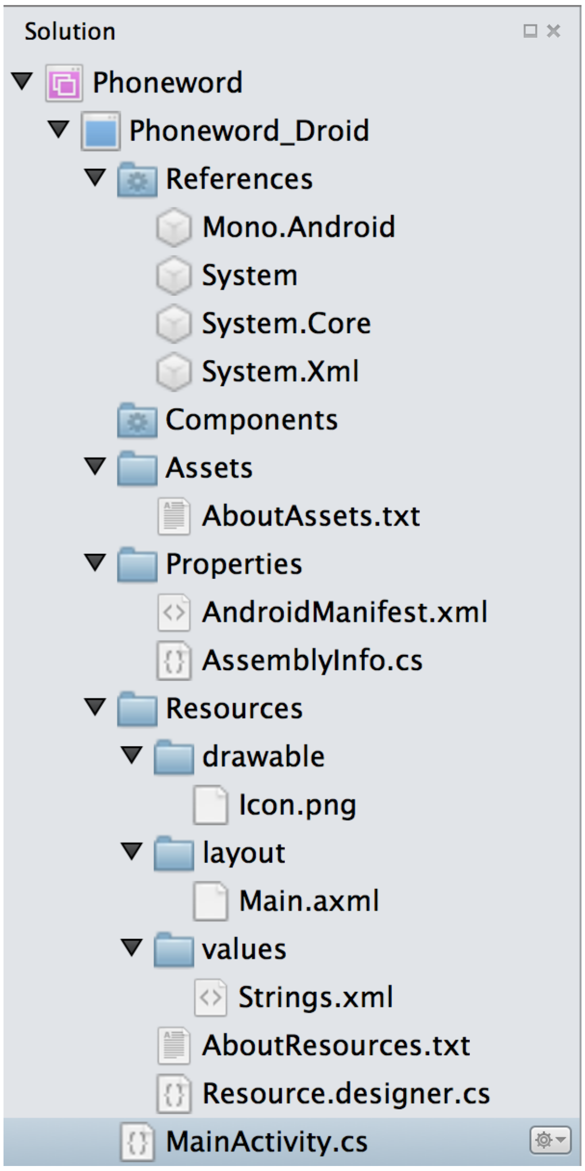
Visual Studio organizes code into *Solutions* and *Projects*. A Solution is a container that can hold one or more Projects. A Project can be an application (such as iOS or Android), a supporting library, a test application, and more. In our Phoneword app, we added a new Android Project using the *Android Application* template to the Phoneword Solution we created in the Hello, Android guide. Our Solution looked like this:



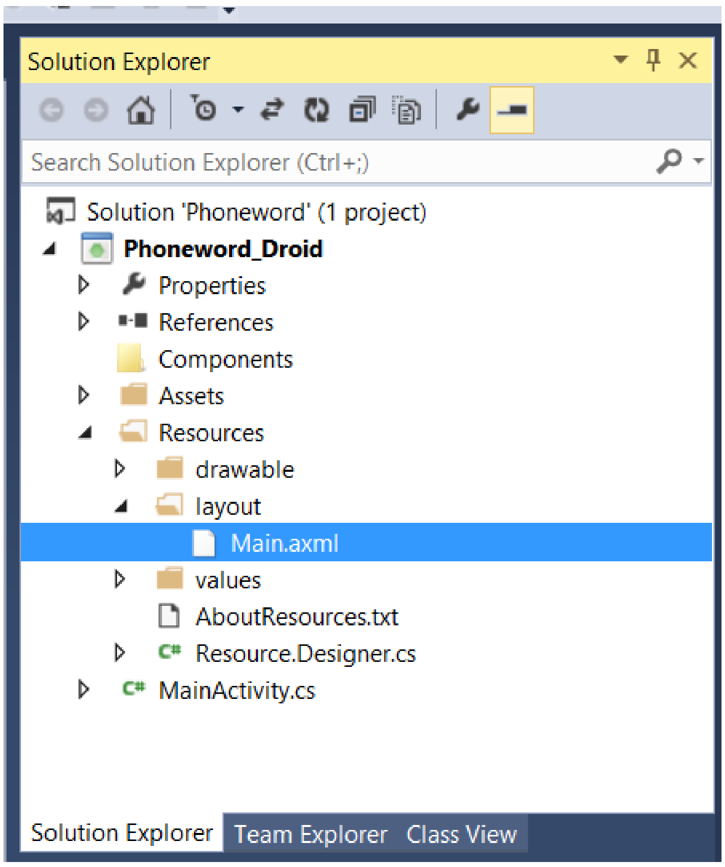
```
using System;

using Android.App;
using Android.Content;
using Android.Runtime;
using Android.Views;
using Android.Widget;
using Android.OS;

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

        protected override void OnCreate(Bundle bundle)
        {
            base.OnCreate(bundle);

            // Set our view from the "main" layout resource
            SetContentView(Resource.Layout.Main);

            // Get our button from the layout resource,
            // and attach an event to it
            Button button = FindViewById<Button>(Resource.Id.MyButton);

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

Anatomy of a Xamarin.Android Application

Let’s get oriented with our Solution’s contents. On the left is the Solution Pad, which contains the directory structure and all the files associated with our Solution:



Let's get oriented with our Solution's contents. On the right is the Solution Explorer, which contains the directory structure and all the files associated with our Solution:



We've created a Solution called *Phoneword* and placed our Android project – *Phoneword_Droid* – inside it. Let's take a look at the items inside the Project:

Folder	Purpose
References	Contains the assemblies required to build and run the application. If we expand the References directory, we'll see references to .NET assemblies such as System, System.Core, and System.Xml, as well as a reference to Xamarin's Mono.Android assembly.
Components	The Components directory houses ready-made features from the Xamarin Components store, a public marketplace for Xamarin code. For more information on Xamarin Components, refer to the Xamarin Components walkthrough.
Assets	Contains the files the application needs to run including fonts, local data files, and text files. Files included here are accessible through the generated

Folder	Purpose
	`Assets` class. For more information on Android Assets, see the Xamarin Using Android Assets guide.
Properties	Contains the AndroidManifest.xml file that describes all the requirements for our Xamarin.Android application, including name, version number, and permissions. The *Properties* folder also houses AssemblyInfo.cs, a .NET assembly metadata file. It is good practice to fill this file with some basic information about your application.
Resources	Contains application resources such as strings, images, and layouts. We can access these resources in code through the generated `Resource` class. The Android Resources guide provides more details about the *Resources* directory. The application template also includes a concise guide to Resources in the *AboutResources.txt* file.

Resources

The *Resources* directory contains three folders named *drawable*, *layout*, and *values*, as well as a file named *Resource.designer.cs*. The items are summarized in the table below:

Item	Purpose
drawable	The drawable directory houses drawable resources such as images and bitmaps. In the default template, the drawable directory houses the application icon file, `Icon.png`.
layout	The layout directory contains *Android designer files* (.axml) that define the user interface for each screen or Activity. The template creates a default layout called *Main.axml*.

Item	Purpose
values	This directory houses XML files that store simple values such as strings, integers, and colors. The template creates a file to store string values called *Strings.xml*.
Resource.designer.cs	Also known as the `Resource` class, this file is a partial class that holds the unique IDs assigned to each resource. It is automatically created by the Xamarin.Android tools and is regenerated as necessary. This file should not be manually edited, as Xamarin.Android will overwrite any manual changes.

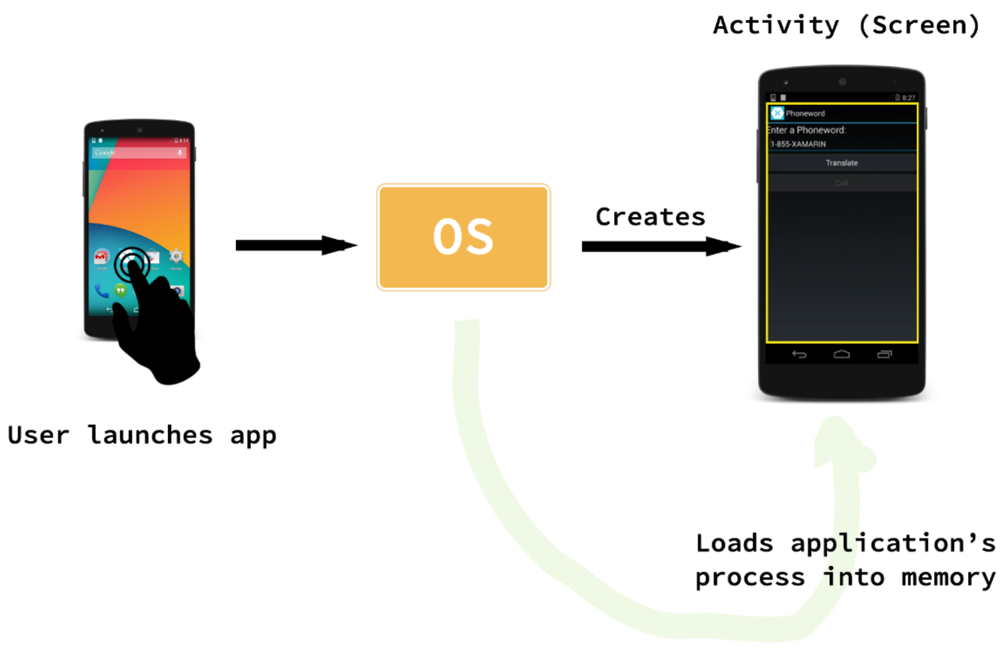
App Fundamentals and Architecture Basics

Android applications don't have a single entry point - there's no single line of code in the application that the operating system calls to start the application. Instead, an application starts when Android creates one of its classes, during which Android loads the entire application's process into memory.

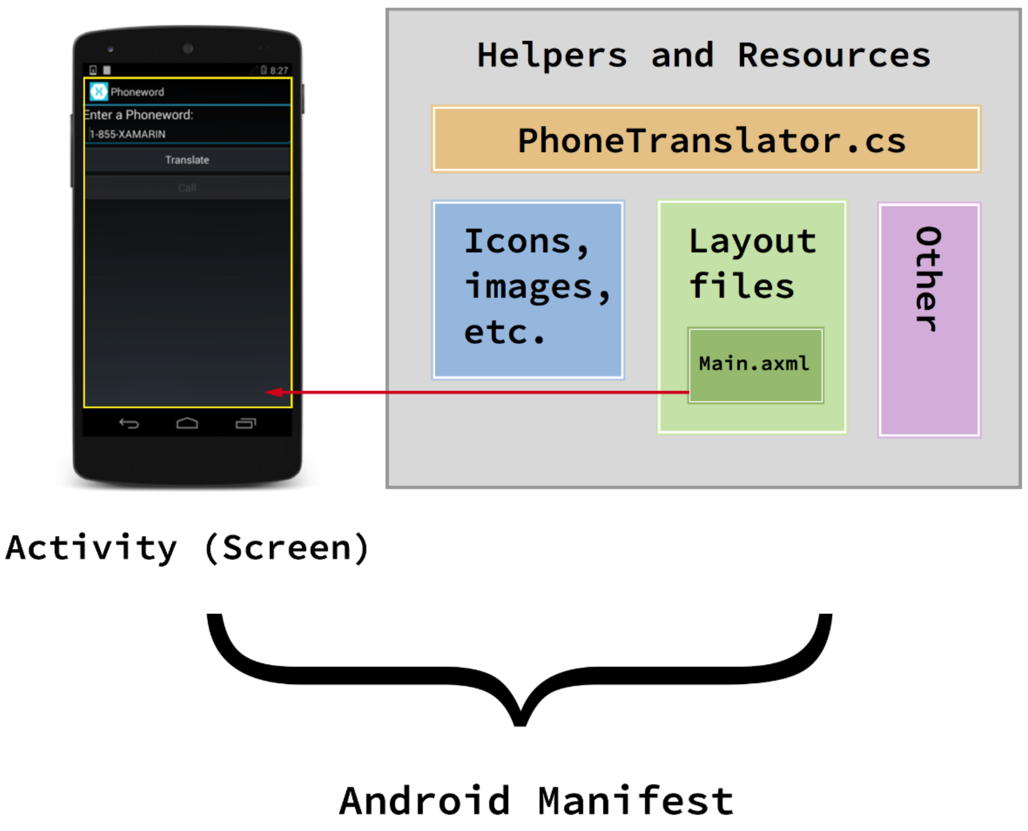
This unique feature of Android can be extremely useful when designing complicated applications or interacting with the Android operating system. However, these options also make Android complex when dealing with a basic scenario like our *Phoneword* application. For this reason, we're going to split our exploration of Android architecture in two. In this guide, we'll dissect an application that uses the most common entry point for an Android app - the first screen. In the Hello, Android Multiscreen guide, we'll dive into the full complexities of Android architecture as we discuss different ways to launch an application.

Phoneword Scenario - Starting with an Activity

When we open our Phoneword application for the first time in an emulator or on a device, the operating system creates the first *Activity*. An Activity is a special Android class that corresponds to a single application screen, and is responsible for drawing and powering the user interface. When Android creates an application's first Activity, it loads the entire application:



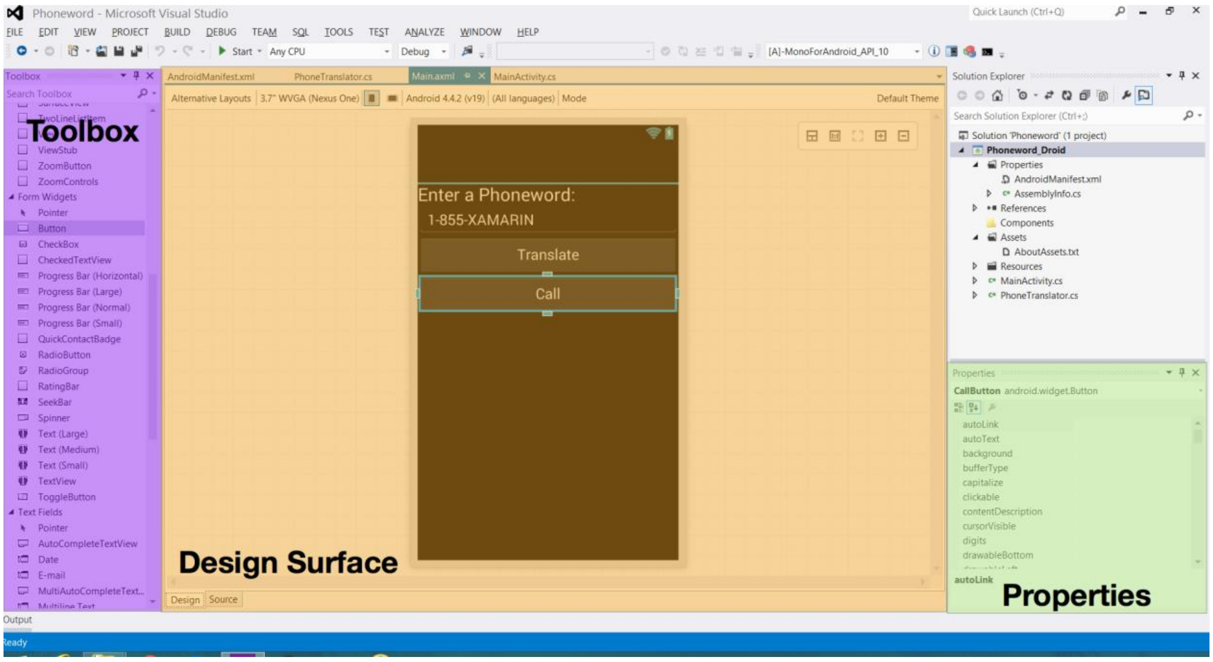
Since there is no linear progression through an Android application – we can launch the application from several points - Android has a unique way of keeping track of what classes and files make up an application. In our Phoneword example, all the parts that make up our application are registered with a special XML file called the *Android Manifest*. The role of the *Android Manifest* is to keep track of an application's contents, properties, and permissions and disclose them to the Android operating system. We can think of our Phoneword application as a single Activity (screen) and a collection of resource and helper files tied together by the Android Manifest file, as illustrated by the diagram below:



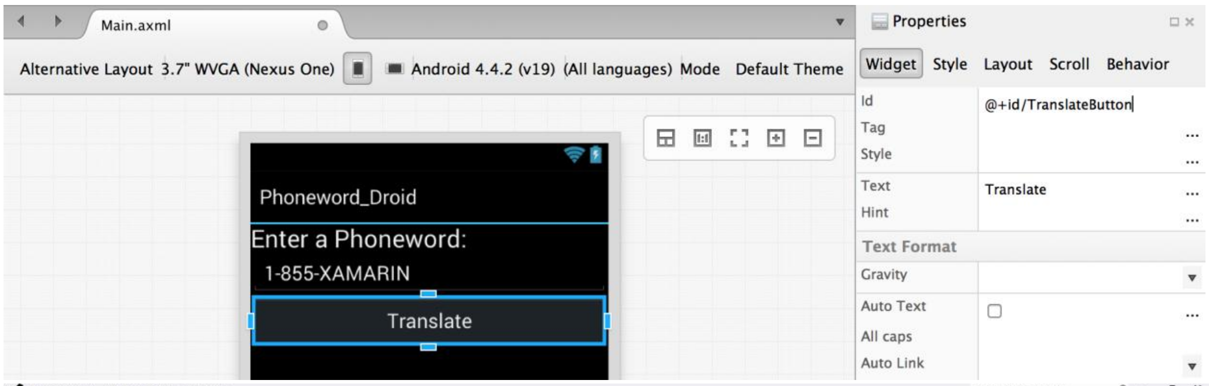
In the next few sections, we'll explore the relationships between the various parts of the Phoneword application to gain a better understanding of the diagram above. We'll begin with the user interface as we discuss the Android designer and layout files.

User Interface

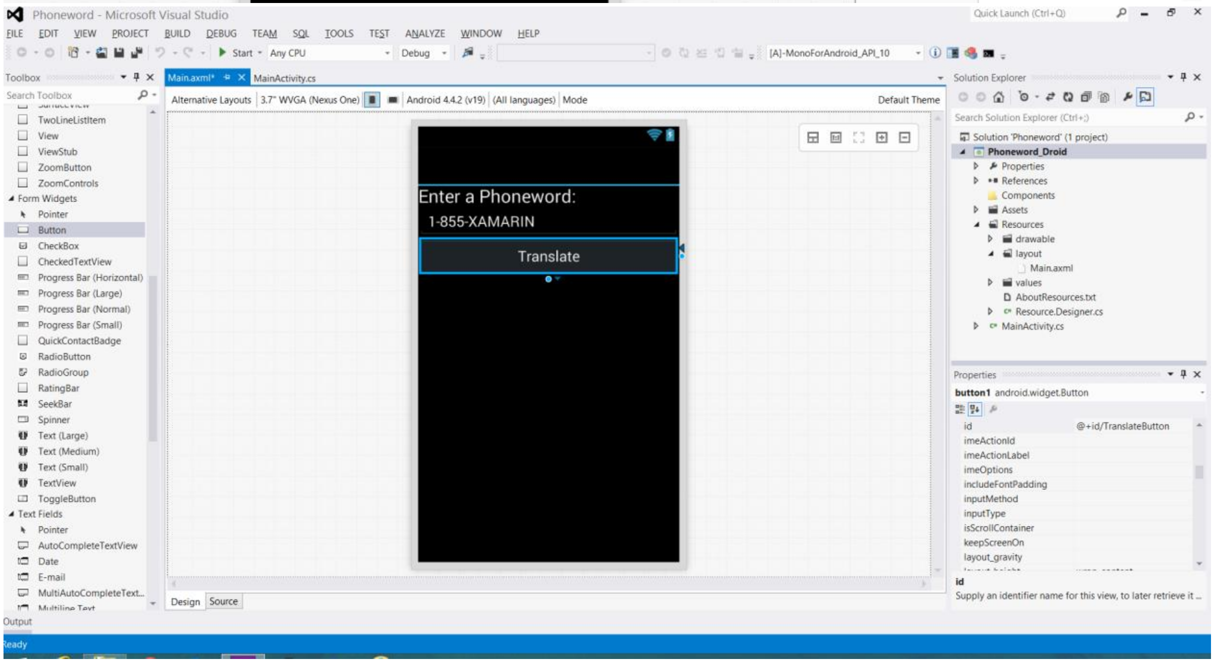
`Main.axml` is the user interface layout file for the first screen in our application. The .axml indicates that this is an Android designer file (AXML stands for *Android XML*). The name *Main* is arbitrary from Android's point of view - we could have easily named the layout file something else. When we open *Main.axml* in our IDE, it brings up the visual editor for Android layout files called the *Android designer*:



In our *Phoneword* app, we set the *TranslateButton*’s ID to `@+id/TranslateButton`:


Main.axml
Alternative Layout 3.7" WVGA (Nexus One) Android 4.4.2 (v19) (All languages) Mode Default Theme
Properties
Widget Style Layout Scroll Behavior
Phoneword_Droid
Enter a Phoneword:
1-855-XAMARIN
Translate

Id	@+id/TranslateButton
Tag	
Style	
Text	Translate
Hint	



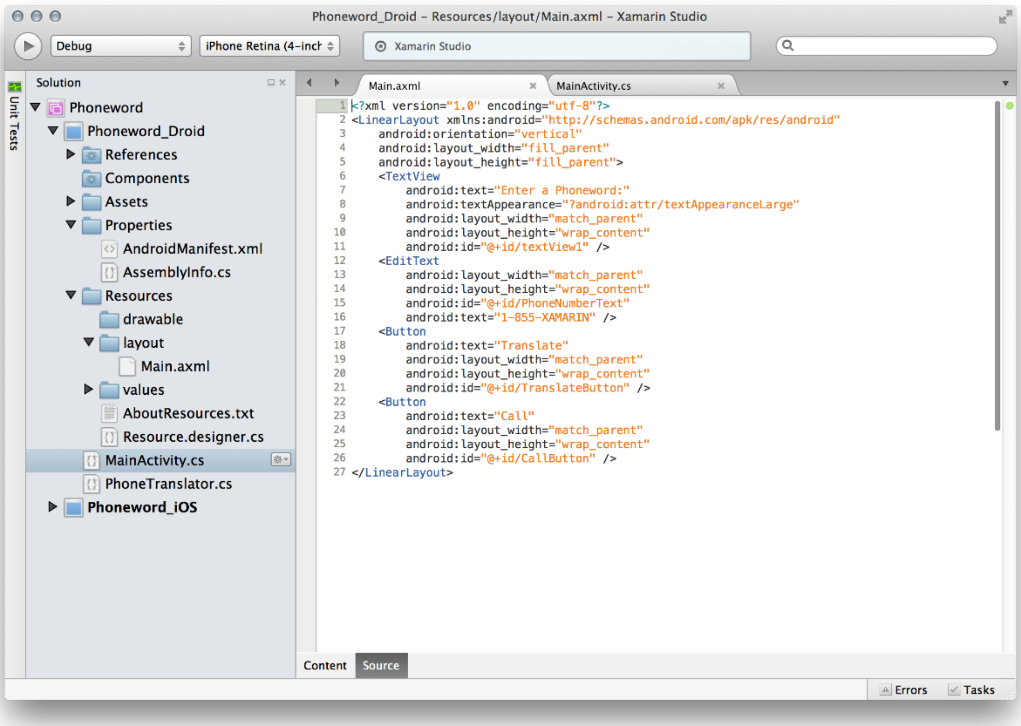
id	@+id/TranslateButton

id
Supply an identifier name for this view, to later retrieve it ...

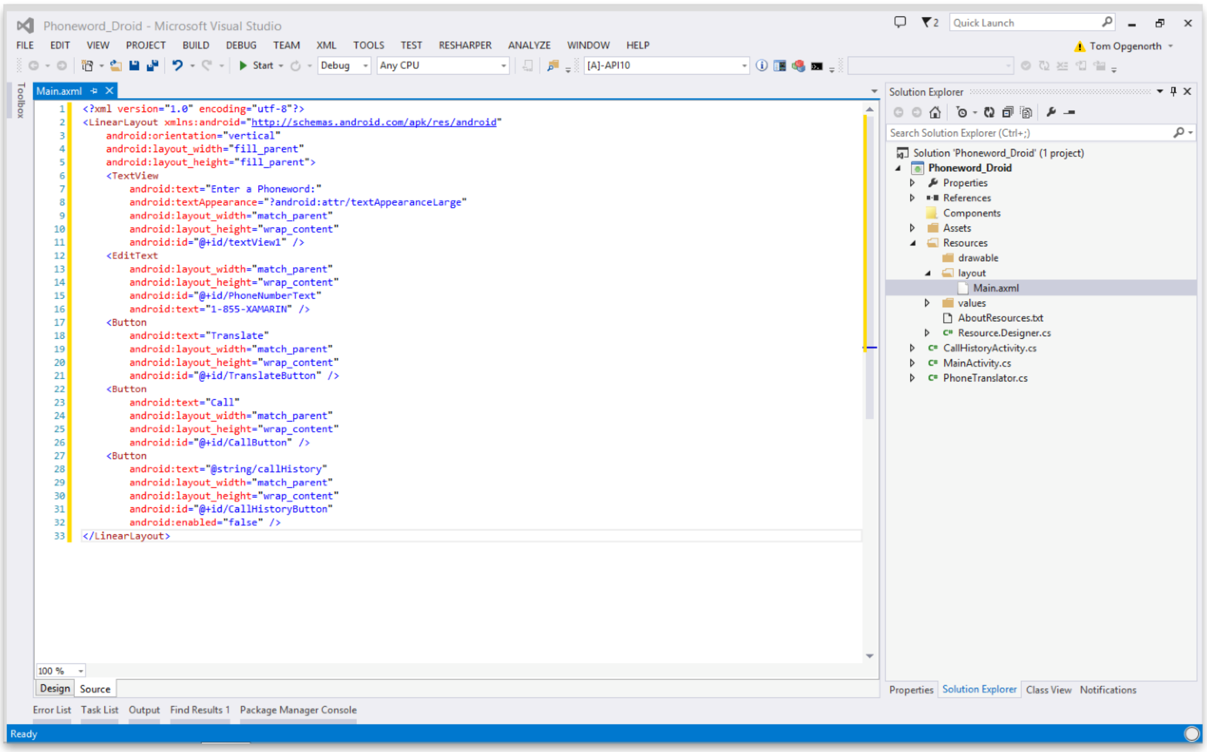
When we set the ID property of the *TranslateButton*, the Android designer maps the *TranslateButton* control to the `Resource` class and assigns it a *resource ID* of `TranslateButton`. This mapping of visual control to class makes it possible for us to locate and use the *TranslateButton* and other controls in our code. We'll cover this in more detail when we break apart the code that powers our controls. All you need to know for now is that the code representation of a control is linked to the visual representation of the control in the designer via the Id property.

Source View

Everything we define on the design surface gets translated into XML for Xamarin.Android to use. The Android designer provides a source view that contains the XML it generated from the visual designer. We can view the XML by switching to the *Source* panel in the lower left of the designer view, as illustrated by the screenshot below:



```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <TextView
        android:text="Enter a Phoneword:"
        android:textAppearance="?android:attr/textAppearanceLarge"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/textView1" />
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/PhoneNumberText"
        android:text="1-855-XAMARIN" />
    <Button
        android:text="Translate"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/TranslateButton" />
    <Button
        android:text="Call"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/CallButton" />
</LinearLayout>
```



```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <TextView
        android:text="Enter a Phoneword:"
        android:textAppearance="?android:attr/textAppearanceLarge"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/textView1" />
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/PhoneNumberText"
        android:text="1-855-XAMARIN" />
    <Button
        android:text="Translate"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/TranslateButton" />
    <Button
        android:text="Call"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/CallButton" />
    <Button
        android:text="@string/callHistory"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/CallHistoryButton"
        android:enabled="false" />
</LinearLayout>
```

The XML source code should contain our *Text (Large)*, *Plain Text*, and two *Button* elements.

For a more in-depth tour of the Android designer, refer to the Xamarin Android Designer Overview guide.

We've covered the tools and concepts behind the visual part of our user interface. Next, let's jump into the code that powers our user interface as we explore Activities and the Activity Lifecycle.

Activities and the Activity Lifecycle

The Activity class contains the code that powers the user interface. The Activity is responsible for responding to user interaction and creating a dynamic user experience.

In this section we introduce the Activity class, discuss the Activity Lifecycle, and dissect the code that powers the user interface in our Phoneword application.

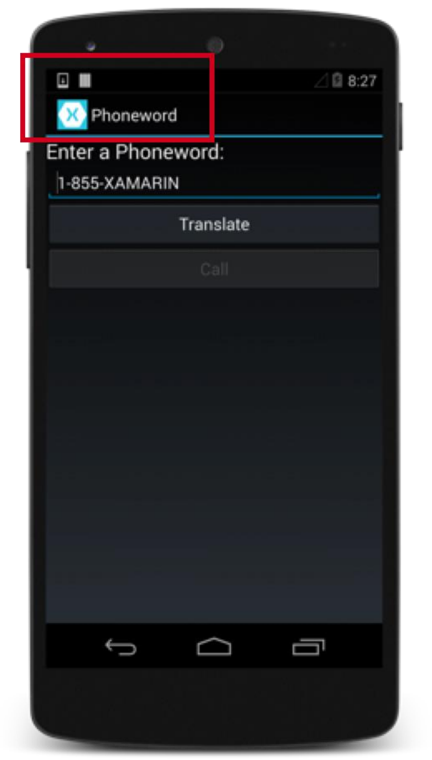
Activity Class

Our Phoneword application has only one screen (Activity). The class that powers the screen is called `MainActivity` and lives in the *MainActivity.cs* file. The name `MainActivity` has no special significance in Android - although it's convention to name the first Activity in an application " `MainActivity`", Android wouldn't care if we named it something else.

If we open the *MainActivity.cs* file, we see that our `MainActivity` class is a ***subclass*** of the `Activity` class, and that the Activity is adorned with the ActivityAttribute:

```
[Activity(Label = "Phoneword", MainLauncher = true, Icon = "@drawable/icon")]

public class MainActivity : Activity

{

...

}
```

The Activity Attribute registers the Activity with the Android Manifest, letting Android know that this class is part of the Phoneword application managed by this Manifest. The `Label` property sets the text to display at the top of the screen, and the `Icon` sets the image to display next to the text, as illustrated by the screenshot below:

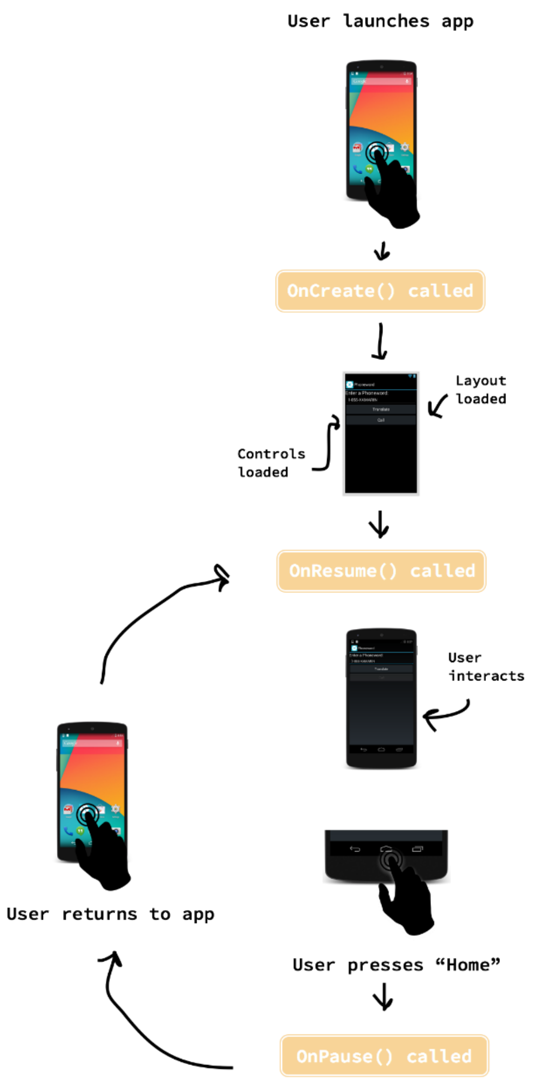


The `MainLauncher` property tells Android to display this Activity when the application starts up. This property becomes important as we add more Activities (screens) to our application in the Hello, Android Multiscreen guide.

Now that we understand the basics of the `MainActivity`, let's dive deeper into the Activity code by introducing the Activity lifecycle.

Activity Lifecycle

In Android, Activities go through different stages of a *lifecycle* depending on their interactions with the user. Activities can be created, started and paused, resumed and destroyed, and so on. The Activity class contains methods that the system calls at certain points in the screen's lifecycle. The following diagram illustrates a typical life of an Activity, and some of the corresponding lifecycle methods:



By overriding Activity lifecycle methods, we can control how the Activity loads, how it reacts to the user, and even what happens after it disappears from the device screen. For example, we can override the lifecycle methods in the diagram above to perform some important tasks:

- **OnCreate** – Create views, initialize variables, and do other prep work before the user sees the Activity. This method is called only once when the Activity is loaded into memory.
- **OnResume** – Perform any tasks that need to happen every time the Activity returns to the device screen.
- **OnPause** – Perform any tasks that need to happen every time the Activity leaves the device screen.

When we add custom code to a lifecycle method in the Activity, we *override* that lifecycle method's *base implementation*. We tap into the existing lifecycle method, which has some code already attached to it, and we extend it with our own code. We call the base implementation from inside our method to make sure the original code runs before our new code. We'll see an example of this in the next section.

The Activity Lifecycle is an important and complex part of Android, and we're not going to dive into the details just yet. If you'd like to learn more about Activities after you finish the Getting Started series, we recommend reading the Activity Lifecycle guide. For now, let's focus on the first stage of the Activity Lifecycle, `OnCreate`.

OnCreate

Android calls the Activity's `OnCreate` method when it creates the Activity, before the screen is presented to the user. We can override the `OnCreate` lifecycle method to create views and prepare our Activity to meet the user:

```
protected override void OnCreate (Bundle bundle)
{
    base.OnCreate (bundle);

    // Set our view from the "main" layout resource
    SetContentView (Resource.Layout.Main);
       // Additional setup code will go here

   }
```

In our Phoneword app, the first thing we do in `OnCreate` is load the user interface we created in the Android designer. To load the UI, we call `SetContentView` and pass it the *resource layout name* for the layout file - our `Main.axml`. Our layout is located at `Resource.Layout.Main`:

```
SetContentView (Resource.Layout.Main);
```

When the `MainActivity` starts up, it will create a view based on the contents of the `Main.axml` file. Note that we've matched our layout name with our Activity name - ***Main***.axml is the layout for ***Main***Activity. This isn't required from Android's point of view, but as we begin to add more screens to the application, we'll find this naming convention makes it easy for us to match the code file to the layout file.

Once the layout file is set, we can start looking up our controls. To look up a control, we call `FindViewById` and pass in the resource ID of the control:

```
EditText                              phoneNumberText                              =
FindViewById<EditText>(Resource.Id.PhoneNumberText);

Button                                translateButton                              =
FindViewById<Button>(Resource.Id.TranslateButton);

Button callButton = FindViewById<Button>(Resource.Id.CallButton);
```

Now that we have a reference to the controls in the layout file, we can start programming them to respond to user interaction.

Responding to User Interaction

In Android, the `Click` event listens for the user's touch. In our app, we handled the `Click` event with a lambda, but we could have also used a delegate or a named event handler. Our final *TranslateButton* code resembled the following:

```
translateButton.Click += (object sender, EventArgs e) =>

{

    // Translate user's alphanumeric phone number to numeric

    translatedNumber                                                               =
Core.PhonewordTranslator.ToNumber(phoneNumberText.Text);

    if (String.IsNullOrWhiteSpace(translatedNumber))

    {

        callButton.Text = "Call";

        callButton.Enabled = false;

    }
    else
    {
        callButton.Text = "Call " + translatedNumber;
        callButton.Enabled = true;
    }
};
```

Additional Concepts Introduced in Phoneword

The Phoneword application introduced several concepts not covered in this guide. These concepts include:

- **Change Button Text** – We learned to change the text of a *Button* by editing that *Button's Text* property. For example, the following code changes the `callButton`'s text to "`Call`":

```
callButton.Text = "Call";
```

- **Enable and Disable Buttons** – *Buttons* can be in an `Enabled` or `Disabled` state. A disabled *Button* won't respond to user input. For example, the following code disables the `callButton`:

```
callButton.Enabled = false;
```

  For more information on buttons, refer to the Android Buttons guide.

- **Display an Alert Dialog** – When the user presses the Call *Button*, our app shows an *Alert Dialog* with the option to place or cancel a call. To create an Alert Dialog, we used an Alert Dialog *Builder*. Our dialog consists of a `Neutral` button to place a call and a `Negative` button to cancels the call, as illustrated by the code below:

```
var callDialog = new AlertDialog.Builder(this);
callDialog.SetMessage("Call " + translatedNumber + "?");
callDialog.SetNeutralButton("Call",
        delegate
        {
            // Create intent to dial phone
        });
callDialog.SetNegativeButton("Cancel", delegate {});


// Show the alert dialog to the user and wait for response.
callDialog.Show();
```

- **Launch Phone App with Intent** – When we wired up the `CallButton`, we sent an *Intent* to launch the system phone app, passing in the telephone number we wanted to call, as illustrated by the code sample below:

```
var callIntent = new Intent(Intent.ActionCall);
callIntent.SetData(Android.Net.Uri.Parse("tel:" +
translatedNumber));
StartActivity(callIntent);
```

Intents are covered in more depth in the Hello, Android Multiscreen guide. In that guide, we'll learn to use Intents to launch other Activities (screens) and even other applications.

Testing, Deployment, and Finishing Touches

Both Xamarin Studio and Visual Studio provide many options for testing and deploying an application. This section covers debugging options, demonstrates testing applications on device, and introduces tools for creating custom app icons for different screen densities.

Debugging Tools

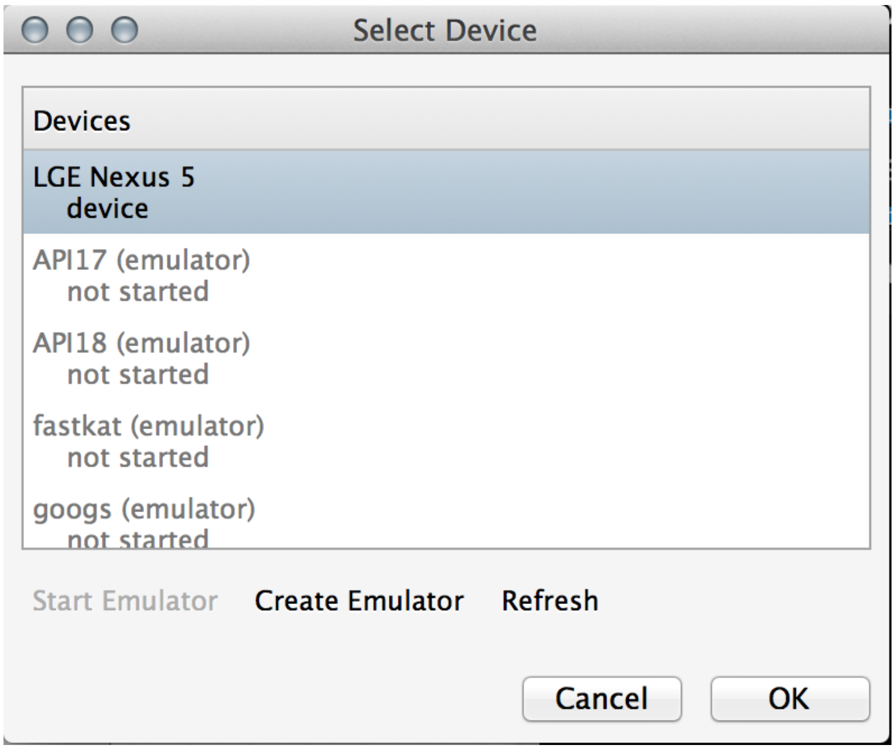
Sometimes issues in application code are difficult to diagnose. To help diagnose complex code issues, we could Set a Breakpoint, Step Through Code, or Output Information to the Log Window.

Deploy to a Device

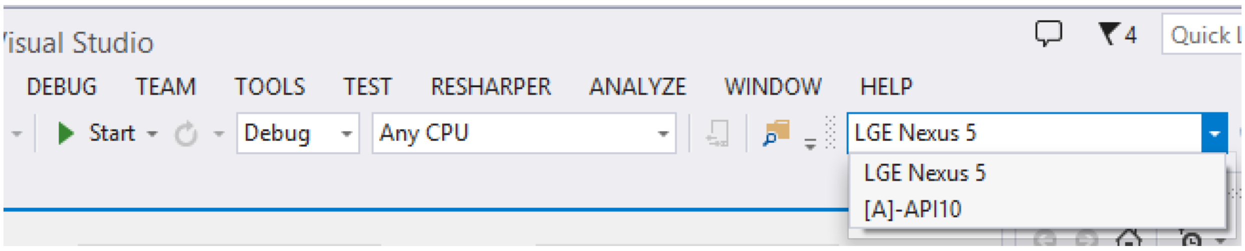
The emulator is a good start for deploying and testing an application, but users will not consume the final app in the emulator. We should test applications on a real device early and often.

Before an Android device can be used for testing applications, it needs to be configured for development. The [Set Up Device for Development]() guide provides thorough instructions on getting a device ready for development.

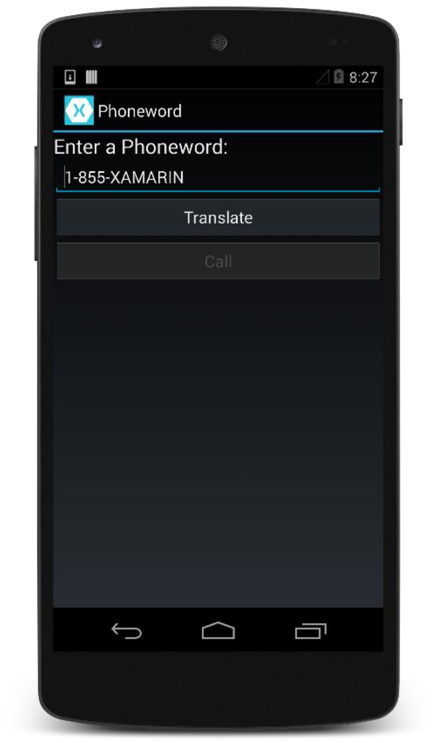
Once the device is configured, we can deploy to it by plugging it in, pressing *Start (Play)*, selecting it from the *Select Device* dialog, and pressing *OK*:



Once the device is configured, we can deploy to it by plugging it in, selecting it from the *Select Device* dialog, and starting the application:

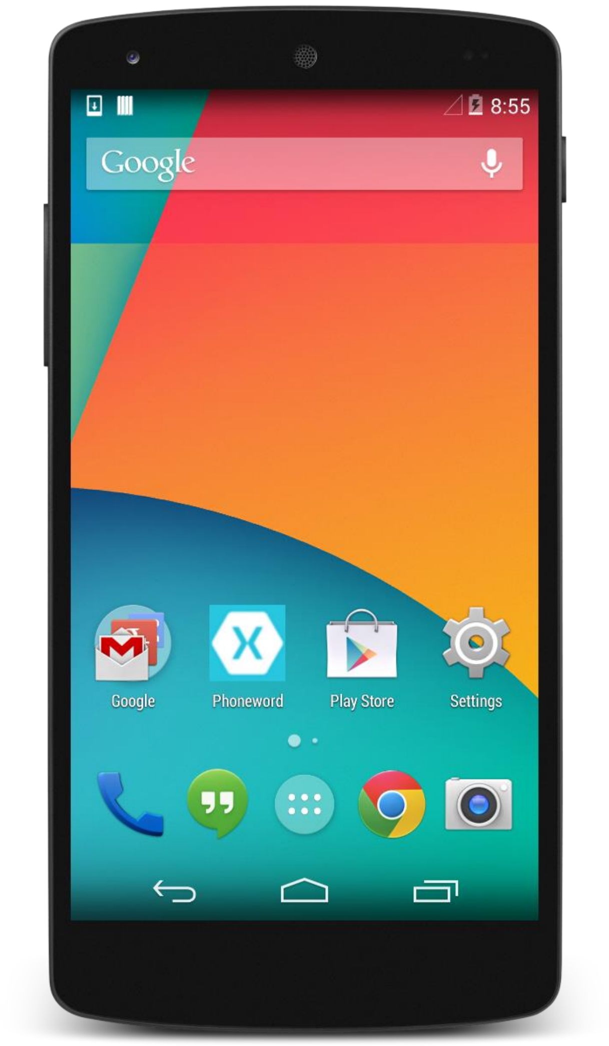


This will launch the application on the device:

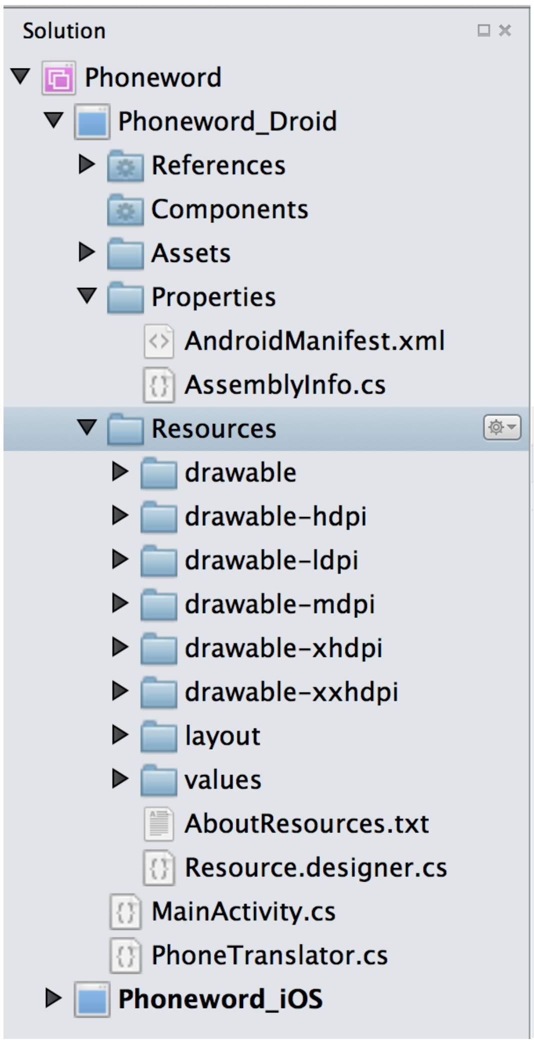


Set Icons for Different Screen Densities

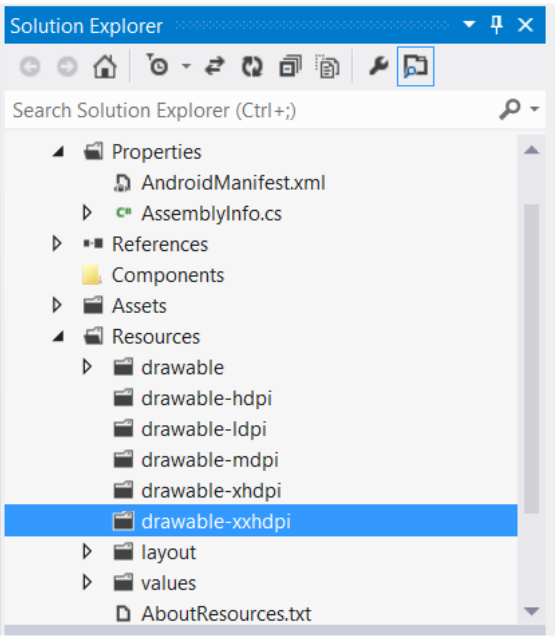
Android devices come in different screen sizes and resolutions, and not all images look good on all screens. For example, here is a screenshot of a low-density icon on a high-density Nexus 5. Notice how blurry it is compared to the surrounding icons:



To account for this, it is good practice to add icons of different resolutions to the Resources folder. Android provides different versions of the drawable folder to handle images of different densities, including an *ldpi* version for low density, *mdpi* for medium, *hdpi* for high, and *xhdpi* and *xxhdpi* for very high density screens. Icons of varying sizes are stored in the appropriate *drawable-** folders:

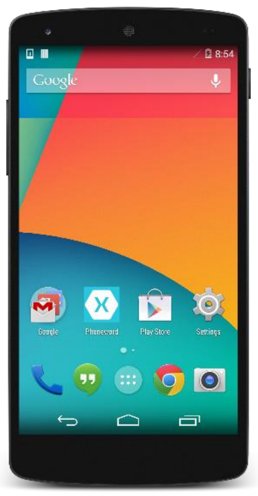


```
Solution
▼ Phoneword
  ▼ Phoneword_Droid
    ▶ References
      Components
    ▶ Assets
    ▼ Properties
        AndroidManifest.xml
        AssemblyInfo.cs
    ▼ Resources
      ▶ drawable
      ▶ drawable-hdpi
      ▶ drawable-ldpi
      ▶ drawable-mdpi
      ▶ drawable-xhdpi
      ▶ drawable-xxhdpi
      ▶ layout
      ▶ values
        AboutResources.txt
        Resource.designer.cs
      MainActivity.cs
      PhoneTranslator.cs
  ▶ Phoneword_iOS
```



```
Solution Explorer
Search Solution Explorer (Ctrl+;)
◢ Properties
    AndroidManifest.xml
  ▷ AssemblyInfo.cs
▷ References
  Components
▷ Assets
◢ Resources
  ▷ drawable
    drawable-hdpi
    drawable-ldpi
    drawable-mdpi
    drawable-xhdpi
    drawable-xxhdpi
  ▷ layout
  ▷ values
    AboutResources.txt
```

Android will pick the icon with the appropriate density:



Generate Custom Icons

Not everyone has a designer available to create the custom icons and launch images an app needs to stand out. Here are several alternate approaches to generating custom app artwork:

- Android Asset Studio – A web-based, in-browser generator for all types of Android icons, with links to other useful community tools. It works best in Google Chrome.
- Sketch 3 – Sketch is a Mac app for designing user interfaces, icons, and more. This is the app we used to design the Xamarin App Icons and Launch Images set. Sketch 3 is available on the App Store and costs about $80. You can try out the free Sketch Tool as well.
- Pixelmator – A versatile image editing app for Mac that costs about $30.
- Visual Studio – You can use this to create a simple icon set for your app directly in the IDE.
- Glyphish – High-quality prebuilt icon sets for free download and purchase.
- Fivrr – Choose from a variety of designers to create an icon set for you, starting at $5. Can be hit or miss but a good resource if you need icons designed on the fly.

For more information about icon sizes and requirements, refer to the Android Resources guide.

Summary

Congratulations! You now have a solid understanding of the components of a Xamarin.Android application as well as the tools required to create it.

In the next tutorial in the Getting Started series, we'll extend our application to handle multiple screens as we explore more advanced Android architecture and concepts.