Hello, Android

Evolve Fundamentals Track, Chapter 4

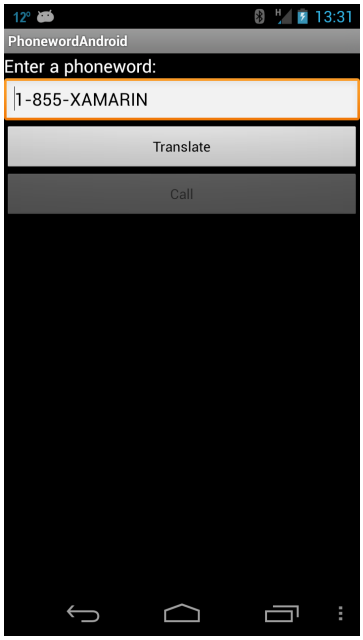
Overview

It is possible to create native Android applications by using Xamarin.Android. Xamarin.Android applications use the same UI concepts and controls as Android but allow you to exploit the power, flexibility, and convenience of C# and the Base Class Library. In addition, Xamarin.Android lets you do this in your choice of IDE and operating system: Xamarin.Studio on OS X or Visual Studio on Windows.

In this chapter we'll look at how to create, deploy, and run a Xamarin.Android application. We will go through the following items:

- Xamarin.Studio – We'll do a quick introduction to the Xamarin IDE. It is just as easy to use Visual Studio 2010 or Visual Studio 2012.
- Anatomy of a Xamarin.Android application – Introduces some of the building blocks of a Xamarin.Android application.
- Android Emulator – Shows how to deploy your application to Android emulator or to a device while you are developing your application.
- Xamarin Designer – Xamarin.Android comes with a powerful WSIYWIG designer that can be used to create user interfaces.

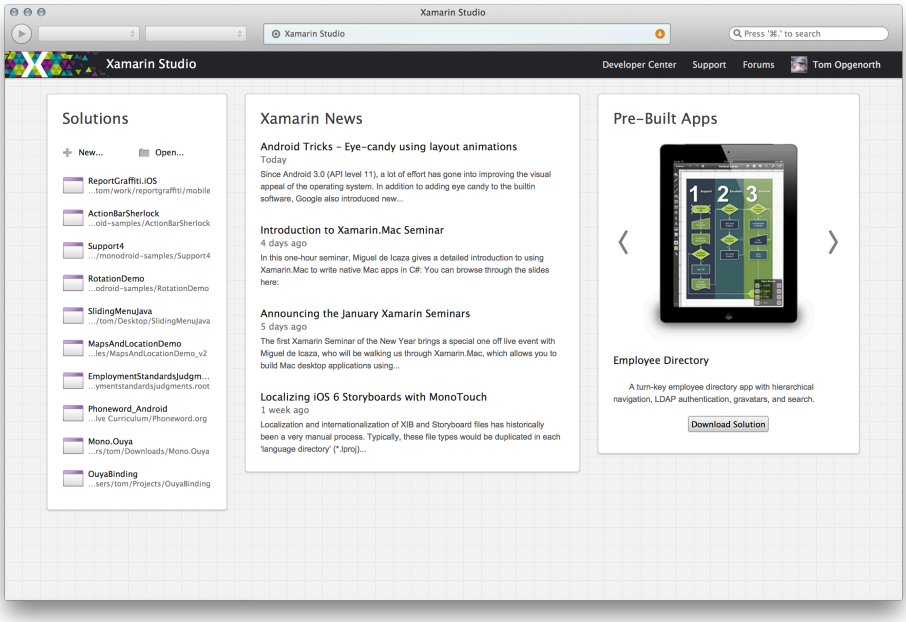
The following screenshot shows an example of the application you will have created by the end of this document:



Introduction to Xamarin.Studio

This document assumes that you are using Xamarin.Studio as your IDE of choice. It is just as easy to use as Visual Studio and, aside from the differences in the screenshots, the same concepts apply to each IDE.

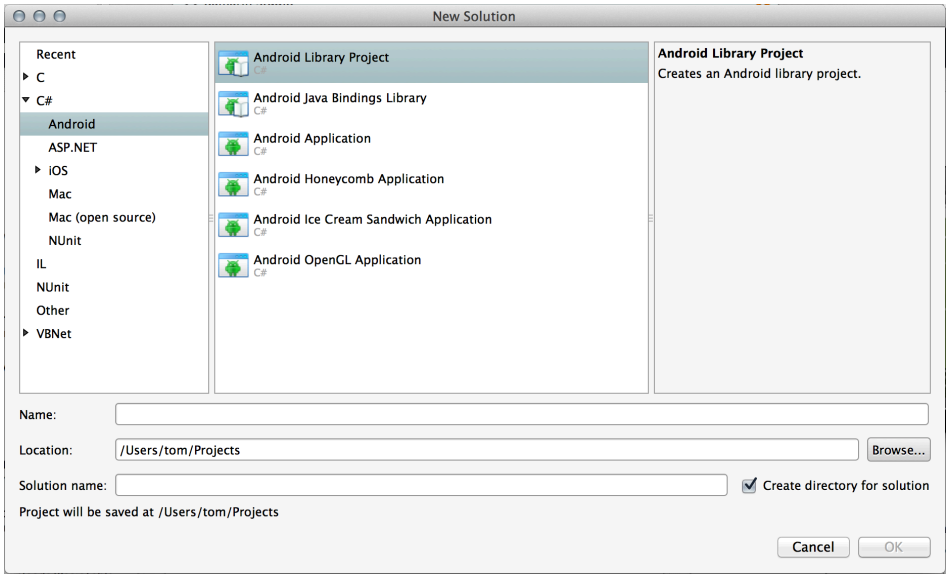
When you launch Xamarin.Studio, you should be presented with a screen that looks something like the following:



Before we dive in and create our Phoneword application, we will first create a very simple Xamarin.Android application so that we can become familiar with Xamarin.Android and the parts of an Android application.

Creating an Android Application

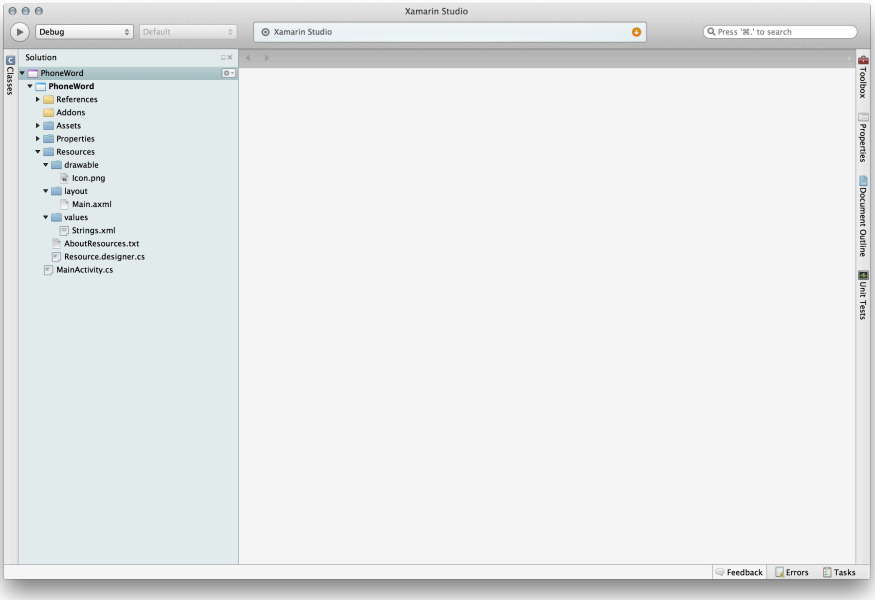
From the Xamarin.Studio Home screen, click **New** in the left pane labeled **Solutions**. This will present you with the following dialog:



This dialog shows the various Xamarin.Android templates for creating projects, including:

- ➔ **Android Library Project** – A reusable .NET library project for Android.
- ➔ **Android Java Bindings Library** – A project for binding an existing JAR file or Android Library project that may be reused in a Xamarin.Android application.
- ➔ **Android Application** – A basic starter project with a single Activity targeting Android 2.2.
- ➔ **Android Honeycomb Application** – A basic starter project with a single Activity targeting Android 3.1.
- ➔ **Android Ice Cream Sandwich Application** – A basic starter project with a single Activity targeting Android 4.0.3.
- ➔ **Android OpenGL Application** – An OpenGL starter project.

In the left-hand pane, choose **C# > Android > Android Application**. This will create a new Xamarin.Android application. Name this new project `PhoneWord`. Select the location where you would like the solution to reside, and then click **OK**. Xamarin.Android will create a new solution that looks something like the following:



The Project

Let's take a look at what has been created. The `PhoneWord` project created three folders that are common to all Xamarin.Android applications. These folders are named **Assets**, **Properties**, and **Resources**. These items are summarized in the table below:

Folder	Purpose

Assets	Contains any type of file the application needs included in its package. Files included here are accessible at runtime via the `Assets` class. These files may be organized into a folder hierarchy.
Properties	Contains normal .NET assembly metadata.
Resources	Contains application resources such as strings and images, as well as declarative XML user interface definitions. These resources are accessible through the generated `Resource` class. We'll examine resources in more detail later on.

The project template also created the following files:

- Resources/layout/Main.axml – This is a layout file that defines the user interface for the MainActivity.
- Resources/layout/Strings.xml – This is an XML file that contains string resources that will be used by the application.
- Resources/Resource.designer.cs – This file is generated automatically by Xamarin.Android and contains constants that can be used to access resources programmatically.
- MainActivity.cs – This is the Activity that the application will launch on startup.

Let's take a quick look at each of these files, and what they are used for.

Main.axml

The first file we'll examine is `Main.axml`. This file defines the visual structure of a user interface. It uses a relatively straightforward XML vocabulary to define the layouts. This structure provides a clean separation between how the UI is created and how it will be used by the application. The following XML is an example of a default layout. It contains a root element called a `LinearLayout`, which is a container for a UI control `Button` named `myButton`:

```
<?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">
    <Button
        android:id="@+id/myButton"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello" />
</LinearLayout>
```

In the XML above, notice the special syntax to name the button: `@+id/myButton`. This syntax tells the Android parser to use the supplied name `myButton` and generate a *Resource ID* with that name. A Resource ID is a special unique integer constant that is generated by Android at compile time. The resource ID is used by methods that need to refer to Android resources. These IDs are stored in the file `Resource.designer.cs`, which we will describe in more detail below.

Likewise, notice that the `Text` property of the button is set to the value of `@string/hello`. This is an example of declaratively referencing a string resource that is contained in the file `Resources/values/Strings.xml`. Let's take a look at how string resources are managed in Android.

Strings.xml

Android accesses resources, such as strings, in an unusual way. All Android resources are kept under the Resources folder in the **Solution Explorer**. To make them accessible from code, generated `Resource` classes are updated for all resources that are included in the various Resources subfolders.

The following XML shows an example of a `Resources/Strings.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
        <string name="hello">Hello World, Click Me!</string>
        <string name="app_name">PhoneWord</string>
</resources>
```

Each string element in the file defines one string element as a key-value pair. At compile time, Android will generate a unique resource ID as a constant and store that inside the file `Resources/Resource.designer.cs`. The string element uses the `name` attribute as a key for each element.

These string resources can be accessed programmatically by passing the resource ID as a parameter to various methods, or declaratively by using the `@string/name` syntax.

Resource.designer.cs

Xamarin.Android generates unique integer IDs for resources, and stores these constants in the file `Resources/Resource.designer.cs`. Xamarin.Android does this for anything that is kept in the `Resources` folder. These types of resources include:

- Drawables
- Layouts
- Strings
- Menus
- User interface elements

So, for example, the layout file `Resources/layout/Main.axml` will have a constant `Resource.Layout.Main` that can be passed to the various methods in Android that need to use this layout file. An example of `Resource.designer.cs` can be seen below:

```
#pragma warning disable 1591
```

```
[assembly: Android.Runtime.ResourceDesignerAttribute("PhoneWord.Resource",
IsApplication=true)]

namespace PhoneWord
{

	[System.CodeDom.Compiler.GeneratedCodeAttribute("Novell.MonoDroid.B
uild.Tasks", "1.0.0.0")]
	public partial class Resource
	{

		public static void UpdateIdValues()
		{
		}

		public partial class Attribute
		{

			private Attribute()
			{
			}
		}

		public partial class Drawable
		{

			// aapt resource value: 0x7f020000
			public const int Icon = 2130837504;

			private Drawable()
			{
			}
		}

		public partial class Id
		{

			// aapt resource value: 0x7f050000
			public const int myButton = 2131034112;

			private Id()
			{
			}
		}

		public partial class Layout
		{

			// aapt resource value: 0x7f030000
			public const int Main = 2130903040;

			private Layout()
			{
```

```
                }
            }

            public partial class String
            {

                // aapt resource value: 0x7f040001
                public const int app_name = 2130968577;

                // aapt resource value: 0x7f040000
                public const int hello = 2130968576;

                private String()
                {
                }
            }
        }
    }
    #pragma warning restore 1591
```

Because this file is generated automatically, it is important that you do not make any changes to it. Any manual changes will be lost each time the project is compiled.

MainActivity.cs

The final file that we'll look is `MainActivity.cs`. This file contains the startup Activity for an application. An `Activity` is a class that models a destination where a user can perform some action while using an app, typically via a user interface.

Conceptually, an Activity can be thought of as being mapped to an application screen. An Activity is similar in some ways to a *Page* in *ASP.NET*, in that every Activity has a lifecycle associated with it. The `Activity` class contains methods that the system will call at certain points in the lifecycle. These methods can also be overridden. The `Activity` subclass created by the project template overrides the `OnCreate` method, which is called after an `Activity` first starts. If you want to learn more about Activities and their lifecycle after you finish the Getting Started series, we recommend reading the Activity Lifecycle article. The code in the file `MainActivity.cs` can be seen below:

```
using System;

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

namespace PhoneWord
{
    [Activity (Label = "PhoneWord", MainLauncher = true)]
    public class Activity1 : 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++);
            };
        }
    }
}
```

Let's break this code down line-by-line. Notice that our class is an Activity. Observe that our Activity is adorned with the `ActivityAttribute`. This attribute provides some metadata to the Xamarin.Android build process. The `MainLauncher` property specifies that this is the Activity that should be displayed when the application first starts up, and that the label to display to the Activity is "PhoneWord."

The next thing to notice is that the method `OnCreate` is overridden. This is a lifecycle method that is invoked by the system after Android instantiates this class. We call `SetContentView` and pass it the resource ID for the layout file `Main.axml`. When the Activity starts up, it will create a view based on the contents of this file.

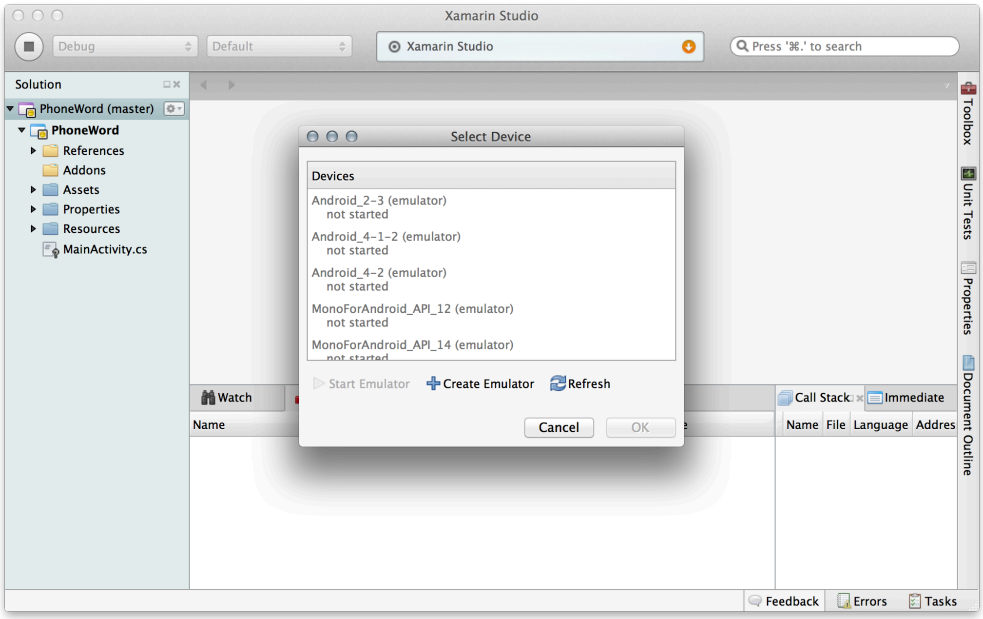
When the user clicks the button, we want to change the text of the button, so let's examine the code for this. First we look up the `Button` instance by using the `FindViewById` method and passing the appropriate resource ID. Then we wire up an event handler to the `Click` event on the `Button` class. The event handler can be an anonymous method, a lambda, or a delegate expression as shown in the following example:

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

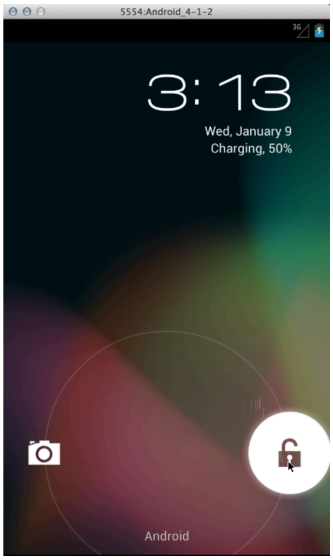
Launching the Emulator - Xamarin Studio

Before we begin to implement our own simple `PhoneWord` app, let's run the application as created from the template. This will let you see a good example of the running app and it will help you become familiar with the process of deploying and launching in the emulator. Under the Xamarin Studio **Run** menu, you have the options to either **Run** or **Debug**. **Debug** will attach the application to the

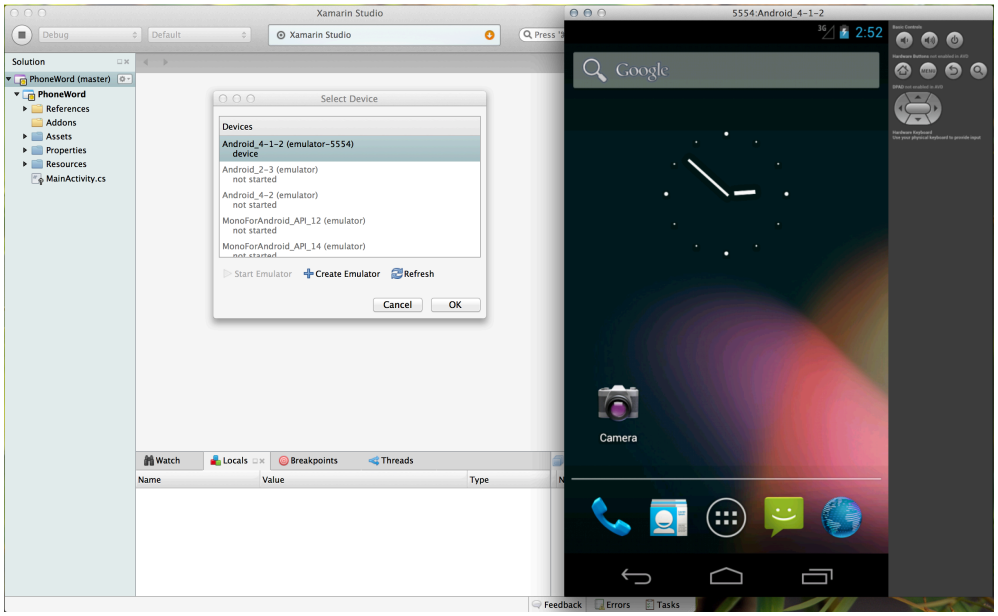
debugger after it launches. For now, let's just select **Run**. This will launch the **Select Device** dialog as shown below:



Xamarin.Studio will take care of launching the emulator for us. Simply select the emulator in the list and choose **Start Emulator**. After the emulator starts up, slide the lock button to the right to unlock and show the Android home screen:



Once the emulator is running, select the emulator in the **Devices** list and click **OK**. The following screenshot shows the emulator running alongside the **Select Devices** dialog:

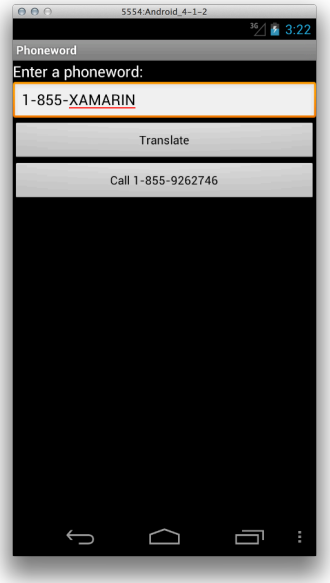


Xamarin.Studio will remember this choice and will try to use this instance of the emulator the next time you run the application. The first time a Xamarin.Android application is installed, the Xamarin.Android shared runtime will be installed, followed by the application. **Installing the runtime only happens for the first Xamarin.Android app deployed to the emulator. It may take a few moments, so please be patient.** Subsequent deployments will only install the app.

The emulator takes a while to launch, so you might consider leaving it running after it starts up. You don't need to shut it down to redeploy your app.

PhoneWord Walkthrough

The default application template we looked at earlier provides a good starting point for creating a simple application. Let's take it a bit further and create a simple application called *PhoneWord*. First, we'll create a small application consisting of a `TextView`, an `EditText`, and two `Buttons`. Entering a phone number that spells a word (such as 1-855-XAMARIN) in the `EditText` and clicking **Translate** will show the phone number and allow the user to dial the phone number by clicking a button. Here is a screenshot that shows the application running in the emulator:



Creating the String Resources

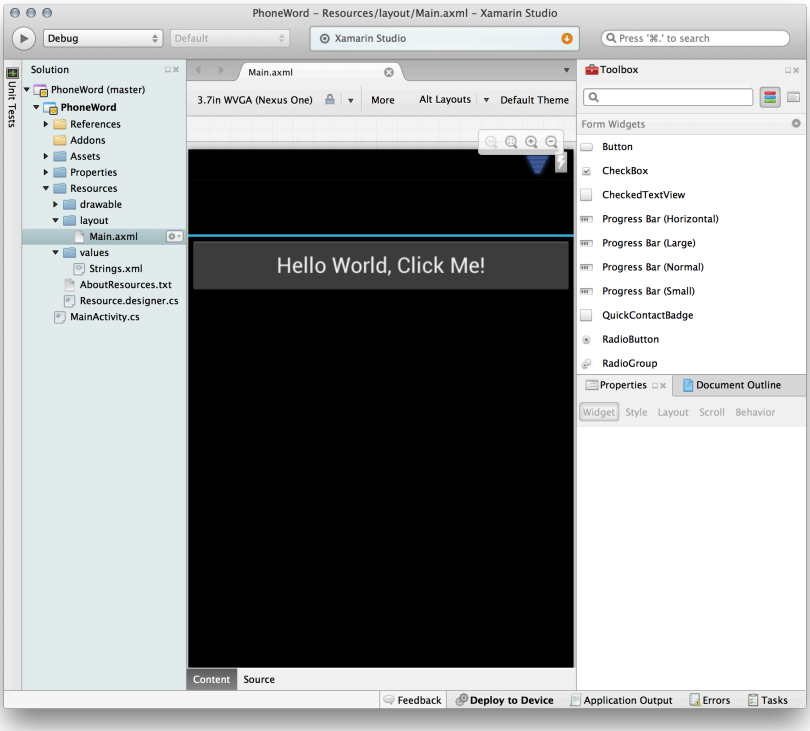
Our application will be more robust if we don't hard code the strings in the layout file. Before we create the layout file, let's create the string resources we need for our application. Edit the file `Resources/Strings.xml` so that it contains the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
	<string name="Enter">Enter a phoneword:</string>
	<string name="Number">1-855-XAMARIN</string>
	<string name="Translate">Translate</string>
	<string name="Call">Call</string>
	<string name="app_name">PhonewordAndroid</string>
</resources>
```

Now we can put together our user interface.

Creating the User Interface

Let's use the Android Designer in Xamarin Studio to create the user interface. Double-click on the file `Resources/layout/Main.axml`, which should start up the designer as shown below:

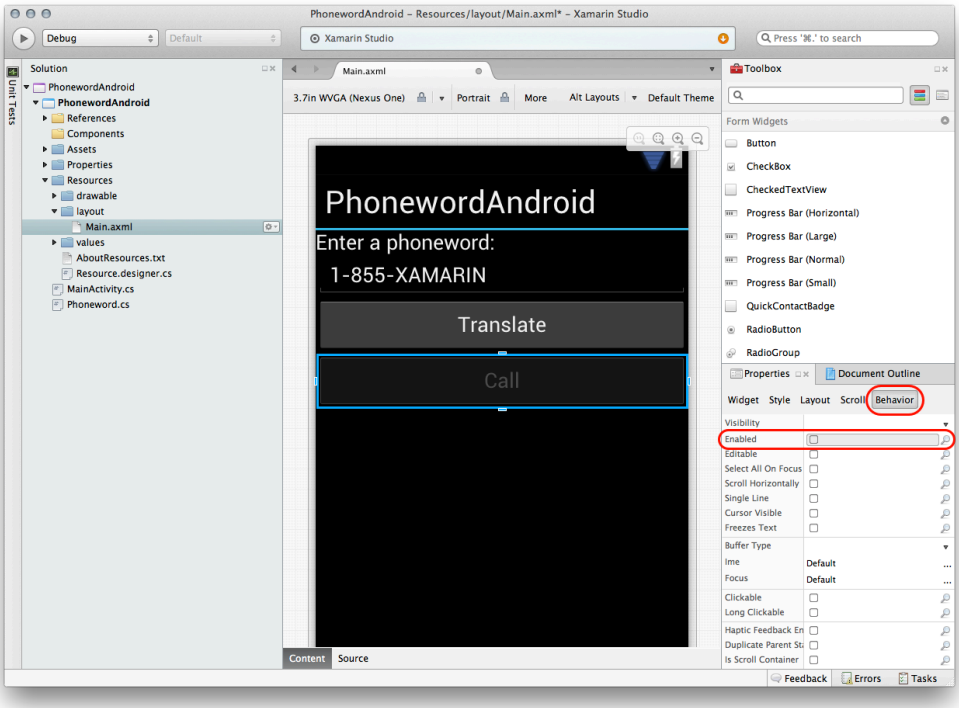


The Xamarin designer is very similar to most drag-and-drop user interfaces. On the right-hand side of the screen, we can see the Toolbox that contains components that we can drag over to the design surface in the middle of the screen. Beneath the Toolbox is the Properties window. The Properties window allows us to manipulate the component we have selected in the design surface.

Let's begin by removing the existing button by selecting and then pressing the Delete key. Next, we need to add the UI elements for our application. Drag the following controls from the Toolbox to the design surface and set the `id` and `Text` properties as shown in the table below:

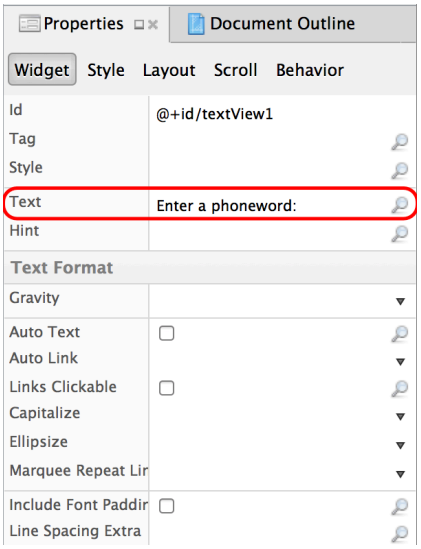
Type of Control	id
`Text (Medium)`	`@+id/Enter`
`PlainText`	`@+id/PhoneNumberText`
`Button`	`@+id/TranslateButton`
`Button`	`@+id/Call`

After adding the `Call` button, make sure that you set the `Enabled` property on the control to `False`. You can do this from the **Properties Pad**, under the **Behavior** tab, as shown in the following screenshot:

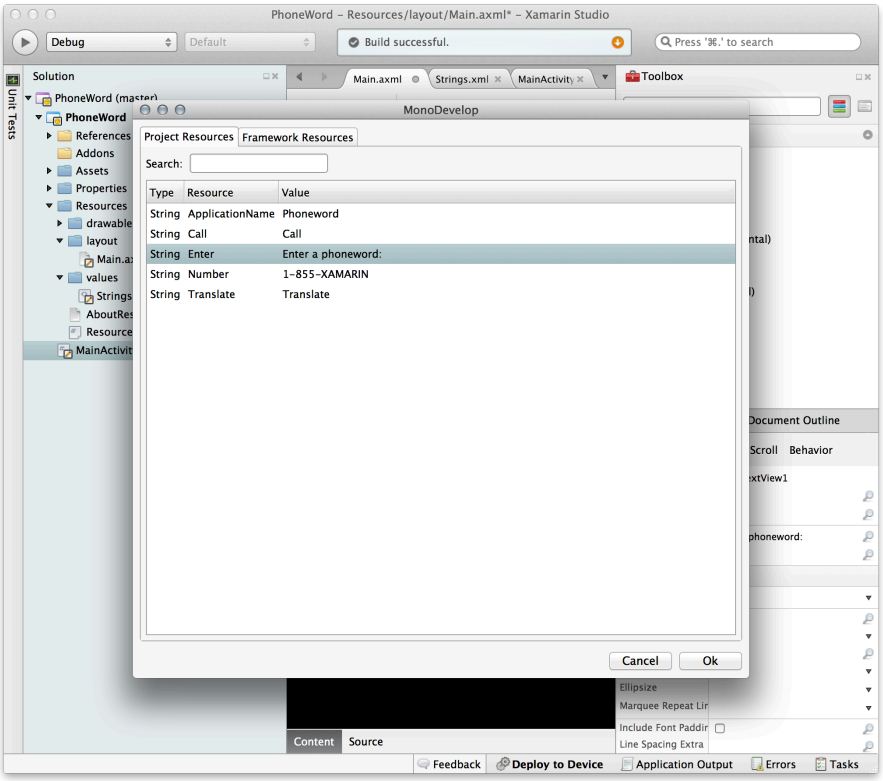


Setting Text Property

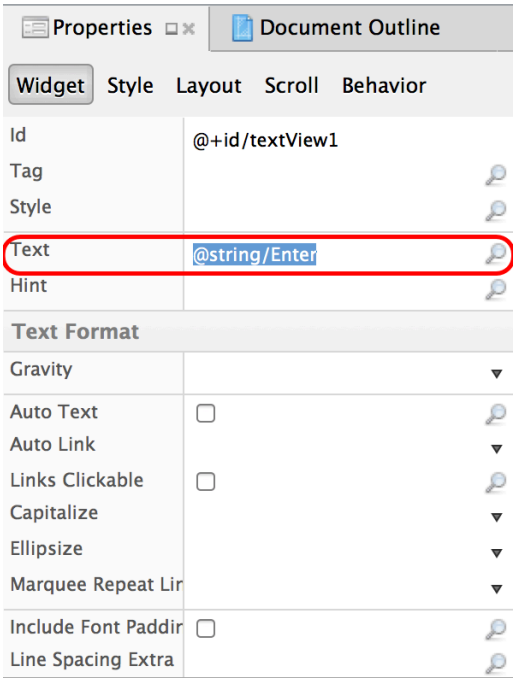
As you add each control to the layout, you can choose one of two ways to set the Text property on each control. One way is to type in the resource by hand. However, it isn't always easy to remember the names of all the string resources. To help with this, Xamarin.Studio provides a dialog to show you the string resources in the application. To use it, click on the magnifying glass that is to the right-hand side of the Properties window, as shown in the following screenshot:



When you do this, a dialog appears that contains a list of all of the string resources in the application, as shown in the following screenshot:



Select the string **Enter**, and click **Ok**. After you do this, notice that the **Text** property of the `TextView` has been changed to `@string/Enter`, as shown in the screenshot below:

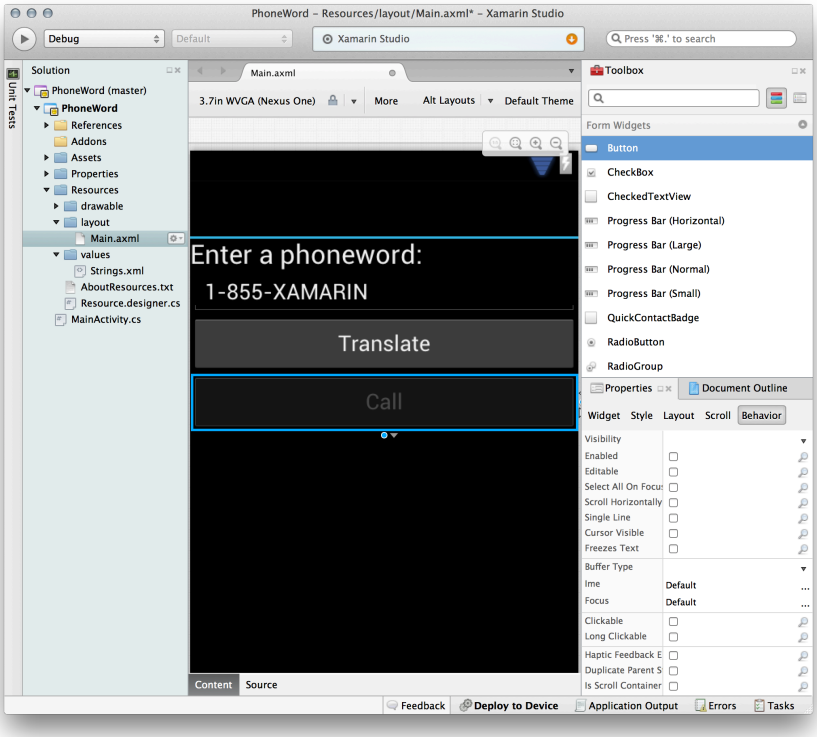


Proceed through each control that you added to the layout, and set the **Text** property on each according to the following table:

Type of Control	id	Text
Text (Medium)	@+id/Enter	@string/Enter
PlainText	@+id/PhoneNumberText	@string/Number
Button	@+id/TranslateButton	@string/Translate
Button	@+id/Call	@string/Call

Checking Our Work So Far

Take a moment to check your work and make sure that you have properly defined your layout. When you're done, the Activity should look something like the following screenshot:



Click **Source** at the bottom of the design surface, next to the **Content** tab. This will show us the XML for this layout, which should look something like the following:

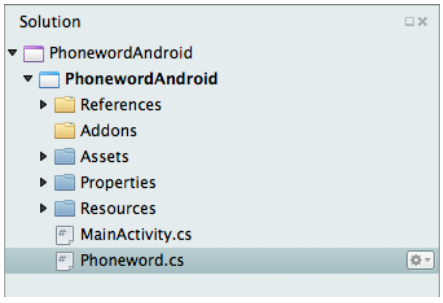
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <TextView
        android:text="@string/Enter"
        android:textAppearance="?android:attr/textAppearanceMedium"
```

```
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:id="@+id/textView1" />
        <EditText
            android:id="@+id/PhoneNumberText"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:text="@string/Number" />
        <Button
            android:id="@+id/TranslateButton"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:text="@string/Translate" />
        <Button
            android:id="@+id/CallButton"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:text="@string/Call"
            android:enabled="false" />
    </LinearLayout>
```

Creating this layout shows how easy and familiar it is for a .NET developer to create user interfaces in Xamarin.Android.

Writing the Code

Now that the user interface of our application is created, we need to write some code and add functionality to the application. To save time, we're going to add the **Phoneword.cs** file from the *Phoneword* project in the companion code. To add it, right-click on the project and choose **Add > Add Files**, then navigate to the **Phoneword.cs** and make sure to choose **Copy** when it asks whether to link or copy. The application should look similar to this in the end:



The next change we need to make is to set up the buttons in our application. Double-click on `MainActivity.cs` so that we may begin editing it. The first thing we need to do is to get rid of the code that was inserted by the Android application template. After our clean up, the file code should look similar to the following:

```
[Activity (Label = "PhonewordAndroid", MainLauncher = true)]
public class Activity1 : Activity
{
        protected override void OnCreate(Bundle savedInstanceState)
        {
```

```
                base.OnCreate(savedInstanceState);
                SetContentView(Resource.Layout.Main);
        }
}
```

We must always call `base.OnCreate(savedInstanceState)` when we override `OnCreate`; this allows the base class to properly initialize the Activity. Next, our code will inflate the layout file that we created earlier and use it for the user interface. To do that, we call `SetContentView` and provide the resource ID to the file `Main.axml`.

Now we need to create the event handlers for the **Translate** button and the **Call** button. Use the following code as a model to update the `OnCreate` method again:

```
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 TranslateButton =
FindViewById<Button>(Resource.Id.TranslateButton);
        Button CallButton = FindViewById<Button>(Resource.Id.Call);
        EditText PhoneNumberText =
FindViewById<EditText>(Resource.Id.PhoneNumberText);

        TranslateButton.Click += delegate
        {

                _translatedNumber =
Core.PhonewordTranslator.ToNumber(PhoneNumberText.Text);

                if (_translatedNumber == "")
                {
                        CallButton.Text =
Resources.GetString(Resource.String.Call);
                        CallButton.Enabled = false;
                }
                else
                {
                        CallButton.Text = Resources.GetString(
                                Resource.String.Call)
                                + " " + _translatedNumber;
                        CallButton.Enabled = true;
                }
        };

        CallButton.Click += delegate
        {
                var callIntent = new Intent(Intent.ActionCall);
                callIntent.SetData(Android.Net.Uri.Parse("tel:" +
                        _translatedNumber));
```

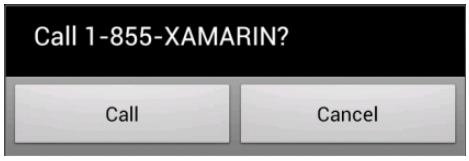
```
                    var callDialog = new AlertDialog.Builder(this)
                            .SetMessage("Call " + PhoneNumberText.Text + "?")
                        .SetNeutralButton("Call", delegate {
                            StartActivity(callIntent);})
                            .SetNegativeButton("Cancel", delegate {});
                    callDialog.Show();
                };
        }
```

Let's look at this code in more detail. The first thing we've done here is to obtain references to the two `Button` widgets and the `TextView` widgets in our UI by making a call to `FindViewById`, and then passing the resource IDs for each control.

Next, we provided an event handler to the `Click` event on the **Translate** button. When the user clicks this button, we make a call to `PhoneWordTranslator.ToNumber` to change the letters to numbers. Then we display the phone number in the **Call** button, and enable or disable it appropriately.

Placing the Call

Let's examine how the Activity makes a phone call. Notice that an event handler is assigned to the `Click` event on the **Call** button. When the user clicks this button, we will display a dialog to the user seeking confirmation that they really do want to call the number. Android provides a helper class called `Android.App.AlertDialog` that we use to create the **Alert** dialog. As you can see in the following screenshot, the **AlertDialog** gives the user two choices:



If the user clicks **Cancel**, then the dialog is dismissed and nothing happens.

What happens when the user clicks **Call**, however, is much more interesting. The Phone application invokes the `StartActivity` method, passing an *Intent* as a parameter. An Intent is a special message that holds an abstract description of something to be done along with some optional data. In this example, our Intent holds the phone number we would like to call. Android will receive this Intent, and realize that it is intended for an application that can place phone calls. The Phone application will start up and receive the Intent as a parameter. The Phone application will extract the phone number from the Intent and place the call.

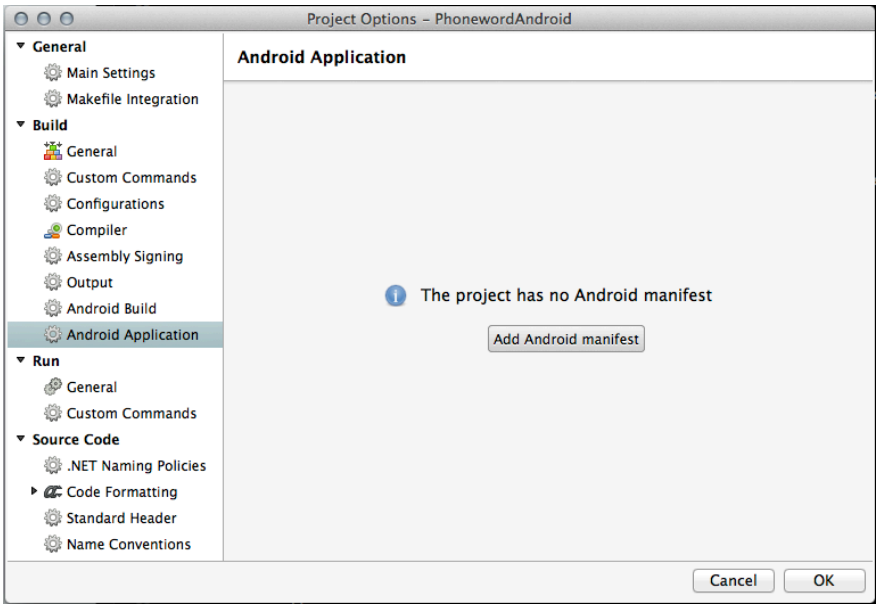
Android Permissions

Although our application is code complete, if we try to run the application and place the call right now, it will crash. The application is missing one crucial thing—permission from the user to place the phone call.

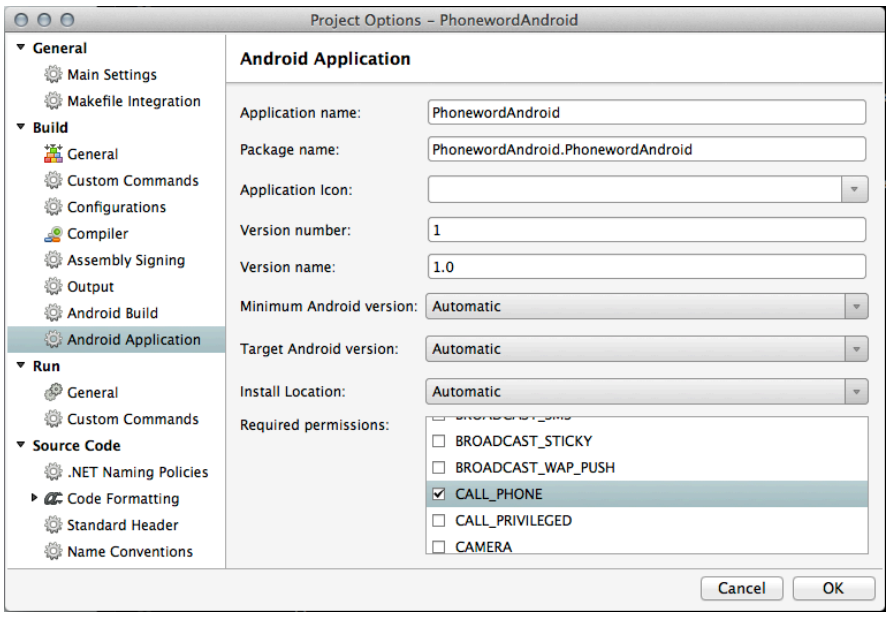
One of the security features of Android is that an application cannot do anything that would affect the user experience or any data on the device. Before an application can do things like access the internet, use the GPS, or place a phone call, it must request permissions from the user. Before a user can install an application, Android will scan the application to see what permissions it is requesting, and ask the user to grant those permissions. These permissions are stored in a special file called the *Android Manifest*.

The Android Manifest is a special XML file that exists in every Android application. It contains metadata about the application that can be used by Android. Permissions are one such example of this metadata. PhoneWord needs to request the permission to phone calls, so let's add those permissions to the application now.

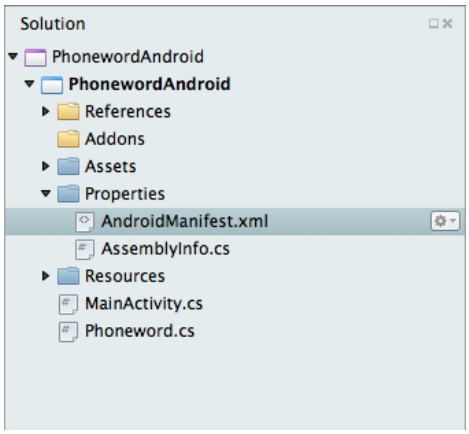
Call up the **Project Options** for PhonewordAndroid, and in the left-hand pane click **Build...Android Application**. The **Project Options** dialog should looking something like the following screenshot:



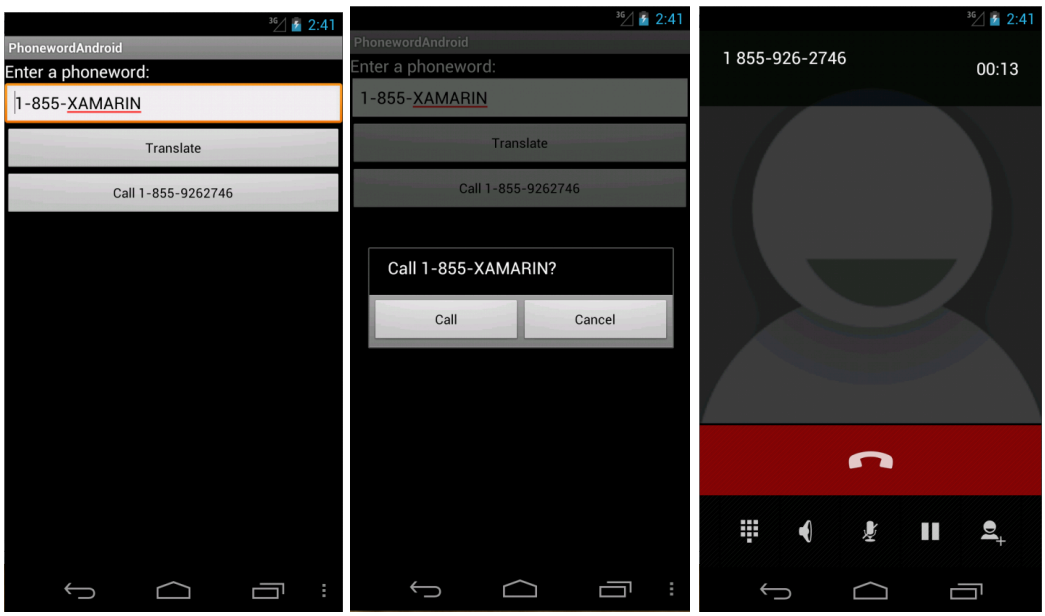
Click **Add Android Manifest**. The dialog should change. A list of **Required permissions** appears at the bottom of the dialog. Scroll through this list and select the box beside the **CALL_PHONE** entry, then click **OK**. The following screenshot shows an example of the **Project Options** dialog as these changes are being made:



Once `OK` has been clicked, Xamarin.Android will add a new file to the project that holds the new permission that was just added. The following screenshot shows this new file, `Properties/AndroidManifest.xml`:



When Mono for Android compiles the application, the contents of this file will be merged into the `AndroidManifest.xml` that will be created for the APK. Now if we run this application, it will be able to place a phone call, as shown in the following screenshots:



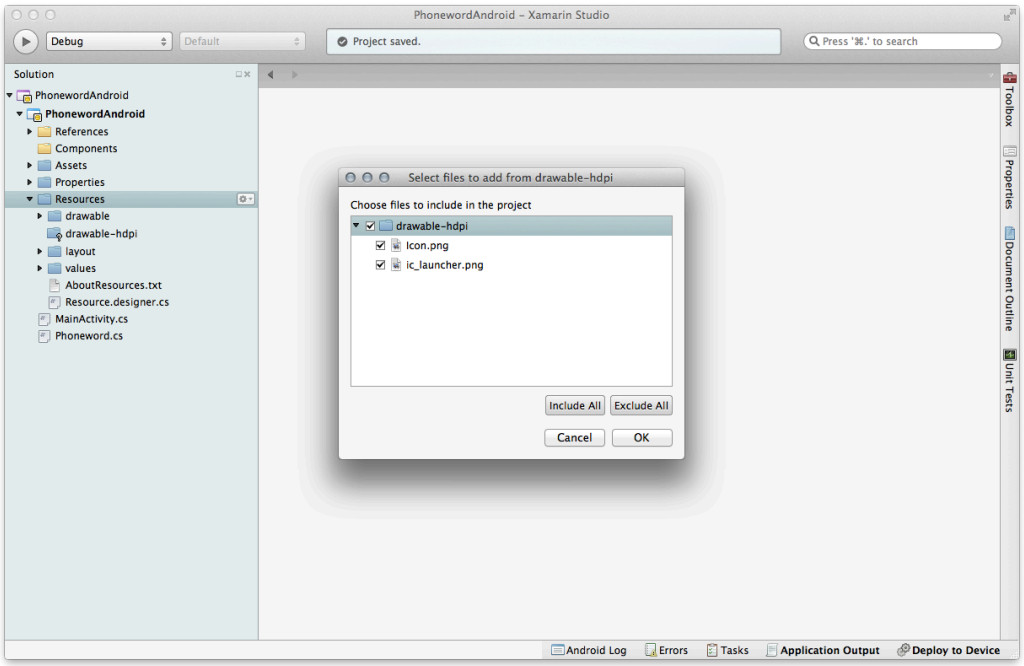
At this point, give yourself a pat on the back. You've written your first Xamarin.Android application! In the next chapter we're going to learn how to create an application with multiple screens, but first, let's take a quick detour to set our application icon, and also learn some more about resources.

Application Icons and Name

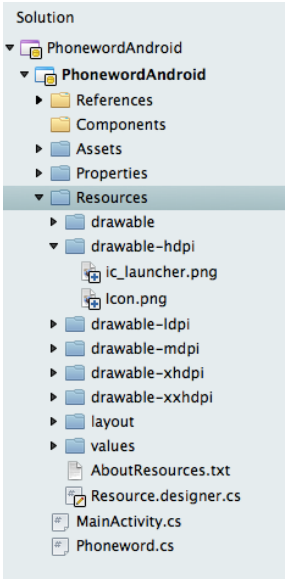
Our application is pretty cool, but if we view it in the Apps list on our Android device, we see that it's been given a default, green Android Robot icon:

Setting Application Icons

We probably want to use an icon that is customized to our app, so let's add one. In the companion code, there is a folder called *App Icons and Splash Screens*, and in there is an *Android* folder. Let's add all the *drawable-** folders (as a copy) in the Android folder into our *Resources* folder of our project. To do this, right-click on the **Resources** folder, and choose **Add Existing Folder**. Do this for each folder. Make sure to include all files:



When finished, our resources folder should look something like this:



Each folder contains icons of different sizes and pixel densities to match different device. We're going to take a closer examination of what that means in a moment, but first, let's add modify the app to use the icon.

Specifying the Activity Icon

Now that we have our Icon (well, many versions of our Icon) added to the project, let's tell Android to use it for our Application. There are actually two places we want to define it: any launchable activities, and also at an application level. Let's modify our Activity to specify the icon by adding an *Icon* parameter to the `Activity` attribute:

```
[Activity (Label = "PhoneWord", MainLauncher = true,
        Icon = "@drawable/ic_launcher")]
public class Activity1 : Activity
{
        ...
```

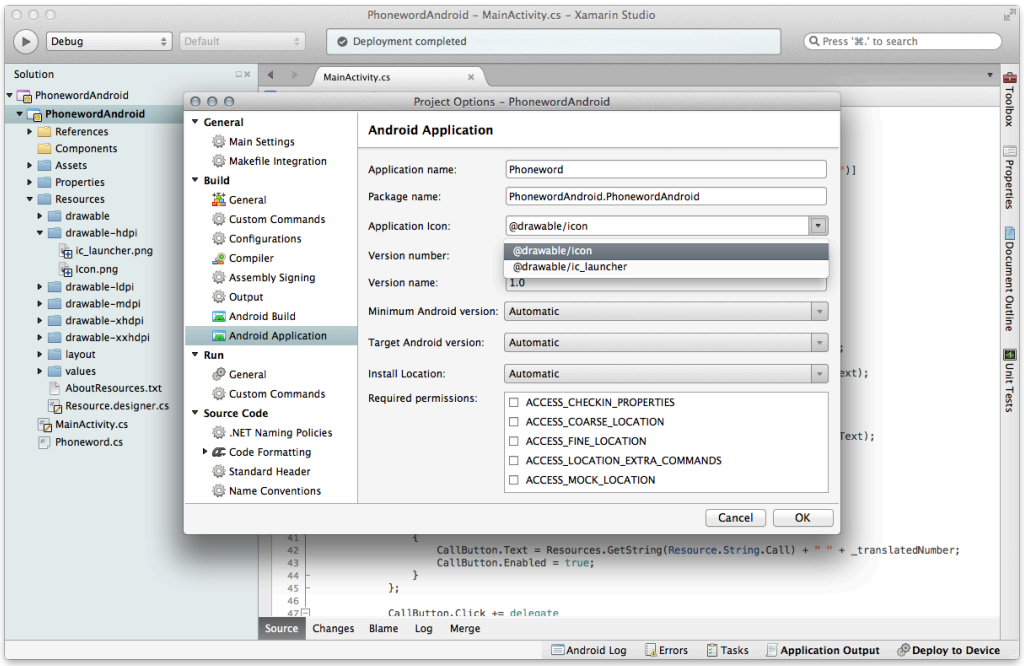
The `Icon` property specifies the icon to use for this Activity. In the example above, Android will use the image located at `Resources/drawable/icon.png` for the icon.

Now, if we deploy the activation to the device we'll see:

Specifying the Application Icon

We've set our Activity icon, but if we look at the Apps installed (available via the *Settings Application*), we still have a default icon:

Set the application icon by double-clicking on the project to open up the properties, then navigate to the **Android Application** dialog, and choose **@drawable/icon** in the **Application Icon** drop down:



The Application Name can also be set here.

If we deploy our app, our custom icon will show up in the settings:

Awesome! Now not only have we built our first application, but we've also spiced it up with custom icons.

We're almost ready to jump into the next chapter on creating multiscreen applications, but first, let's take a look at Resources, which we set aside earlier.

Resources

Android Resources are non-source code files that get compiled into our application and are then available at run time. For instance, layout files, images, icons, videos, audio files, localization strings, etc., are all resources. They're compiled (along with the source code) during the build process, and then are packaged into the APK file:

We've used a couple different resources in our Phoneword app including our layout file and icons.

Resources offer several advantages to an Android application, including:

- **Code-Separation** – Separates source code from images, strings, menus, animations, colors, etc. Separating resources from source code in this way can help streamline localization.
- **Targeting of multiple devices** – Provides simpler support of different device configurations without requiring code changes.
- **Compile-time Checking** – Allows resource usage to be checked at compile time, when it is easier to catch and correct mistakes, as opposed to runtime when it is more difficult to locate errors and costlier to correct them. This is possible because resources are static and are compiled into the application.

There are two ways to access these resources in a Xamarin.Android application: *programmatically* in code and *declaratively* in XML, using a special XML syntax.

We're only going to take a cursory examination of Android Resources here, for a more complete discussion, see the Android Resources guide on the Xamarin Developer Center.

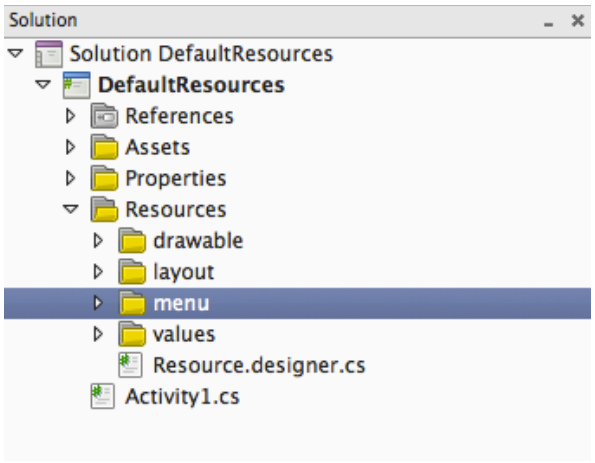
Different Resource Types

One of the challenges of developing Android applications is the vast array of devices that are out there. Android devices come in many different form factors, screen sizes, screen ratios, and even screen densities. If we used the same images and layouts for each one, the user experience could really suffer. As such, a once size fits all approach isn't really appropriate.

For this reason, resources fall into two categories. The first is known as *default resources* and are used by all devices, unless a more specific match is specified. Additionally, all types of resources may optionally have *alternate resources* that Android may use to target specific devices, form factors, or even current language. For example, some applications may provide resources for a different language, such as German. At runtime, Android will detect the locale that the device is set to. If the locale is a German speaking on, then the German resources will be load. If not, then the default resources will be loaded.

Default Resources

Default resources are items that are not specific to any particular device or form factor, and therefore are the default choice by the Android OS if no more specific resources can be found. Because of this, they're the most commonly created type of resource. They're organized according to their resource type, into subdirectories of the Resources directory as shown in the following screenshot:



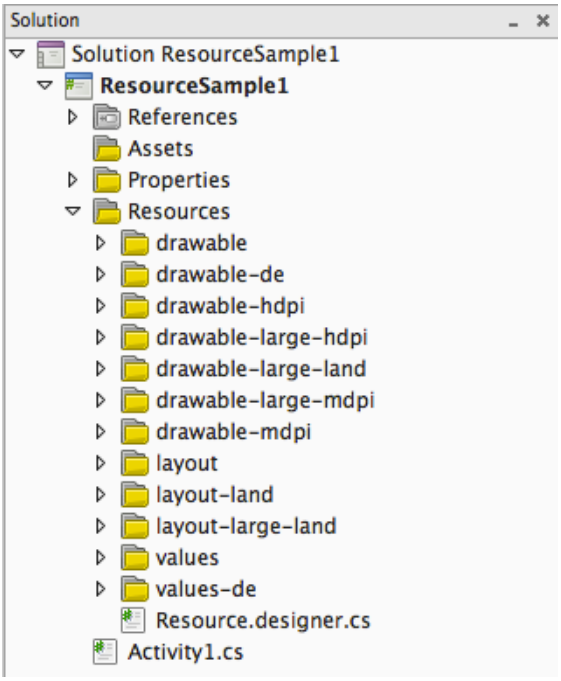
The above screenshot only displays some of the possible resource types in a Xamarin.Android application. For a complete list, see the Android Resources guide.

Alternate Resources

Alternate resources are those resources that target a specific device or run-time configuration, such as the current language, particular screen size, or pixel density. If Android can match a resource that is more specific to a particular device or configuration than the default resource is, then that resource will be used instead. If it does not find an alternate resource that matches the current configuration, then the default resources will be loaded. How Android decides which resources will be used by an application will be covered in more detail below, in the Resource Location section.

Alternate resources are organized, according to the resource type and just like default resources, as a subdirectory inside the Resources folder. The name of the alternate resource subdirectory is in the form `<ResourceType>-<Qualifier>`.

Qualifier is a term that identifies a specific device configuration. There may be more than one qualifier in a name, with each qualifier separated by a dash. For example, the screenshot below shows a project that uses qualifiers and has alternate resources for various configurations such as locale, screen density, screen side, and orientation:



We used qualifiers earlier when we used the prebuilt icons. In our case, we used `hdpi`, `ldpi`, `mdpi`, `xhdpi`, and `xxhdpi`. Each one of these qualifiers specify a different screen density, where `dpi` is an acronym for *Dots Per Inch*, and `h` is high, `l` is low, `m` is medium, and `x` is extra. These five different qualifiers cover nearly every non-television screen density.

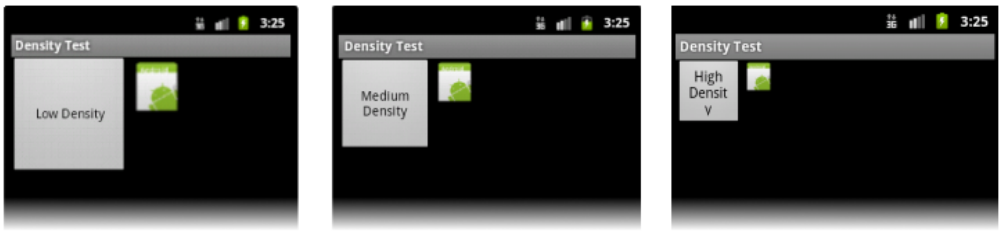
The following rules apply when adding qualifiers to a resource type:

- There may be more than one qualifier, with each qualifier separated by a dash.
- Qualifiers may be specified only once.
- Qualifiers must be specified in the order they appear in the table below.

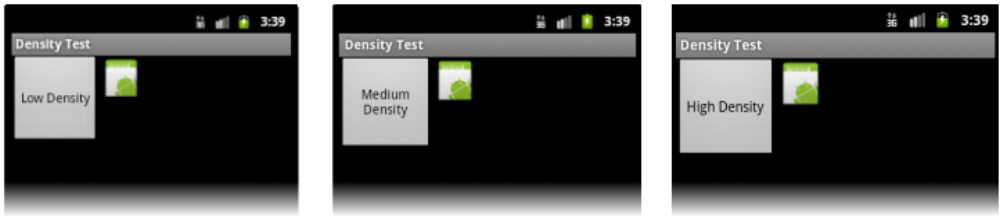
There are a number of qualifiers available beyond just screen density including screen orientation, language, screen size, and device type. For a much more detailed discussion check out the Android Resources guide.

Creating Resources for Varying Screens

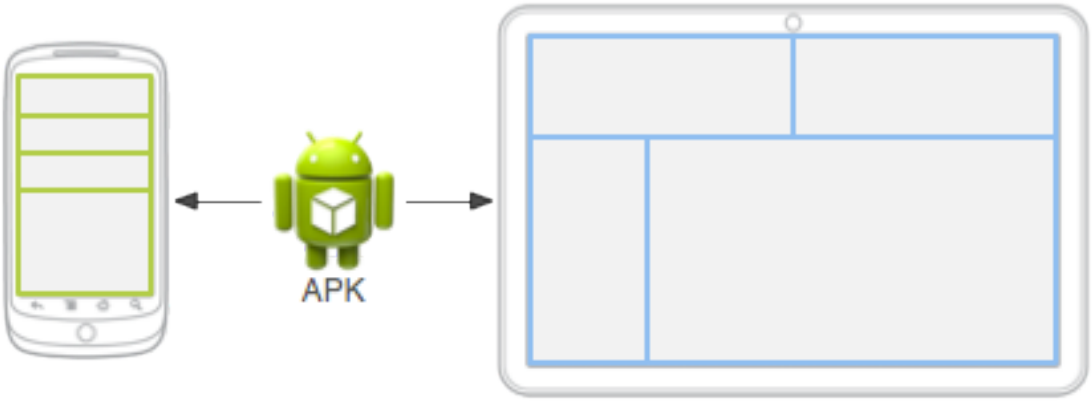
Android itself runs on many different devices and a wide variety of resolutions, screen sizes, and screen densities. Android will perform scaling and resizing that allows applications to work on these devices, but these operations may result in images that will look very shabby on some devices. For example, images may appear blurry, or images may occupy too much (or not enough) screen space, which may cause the position of UI elements in layouts to overlap or be displayed too far apart. The image below shows some layout and appearance problems that occur because density-specific resources were not provided:



Compare this to a layout that is designed with density-specific resources:



To get around this problem, applications need to provide alternate resources that Android can load, depending on the device. This will ensure a consistent looking application regardless of what kind of device it is running on.



Concepts

It's important to understand a few key terms and concepts that are used in relation to screens.

- Screen Size – The amount of physical space available for displaying an application.
- Screen Density – The number of pixels in any given area on the screen. The typical unit of measure is dots per inch (dpi).
- Resolution – The total number of pixels on the screen. When developing applications, resolution is not as important as screen size and density.
- Density-independent pixel (dp) – A virtual unit of measure that allows layouts to be designed independent of density. To convert dp into screen pixels, the following formula is used: *px=dp*dpi/160*.
- Orientation – The current aspect orientation of the device screen. The screen's orientation is considered to be landscape when it is wider than it

is tall. In contrast, portrait orientation refers to a screen that is taller than it is wide. The orientation can change during the lifetime of an application as the user rotates the device.

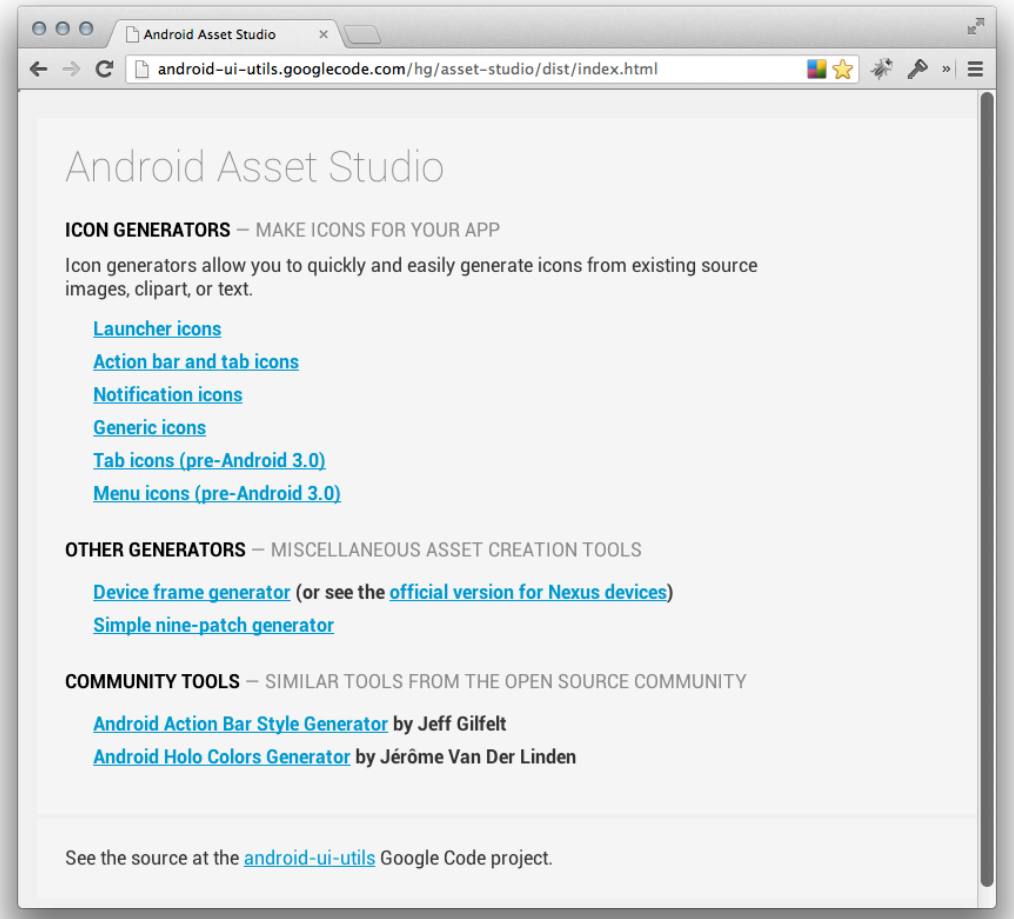
Notice that the first three of these concepts are interrelated; increasing the resolution without increasing the density will increase the screen size. However if both the density and resolution are increased, then the screen size can remain unchanged. This relationship between screen size, density, and resolution complicates screen support.

To help deal with this complexity; the Android framework prefers to use *density-independent pixels (dp)* for screen layouts. By using density-independent pixels, UI elements appear to the user to have the same physical size on screens with different densities (as seen in the images from the previous section).

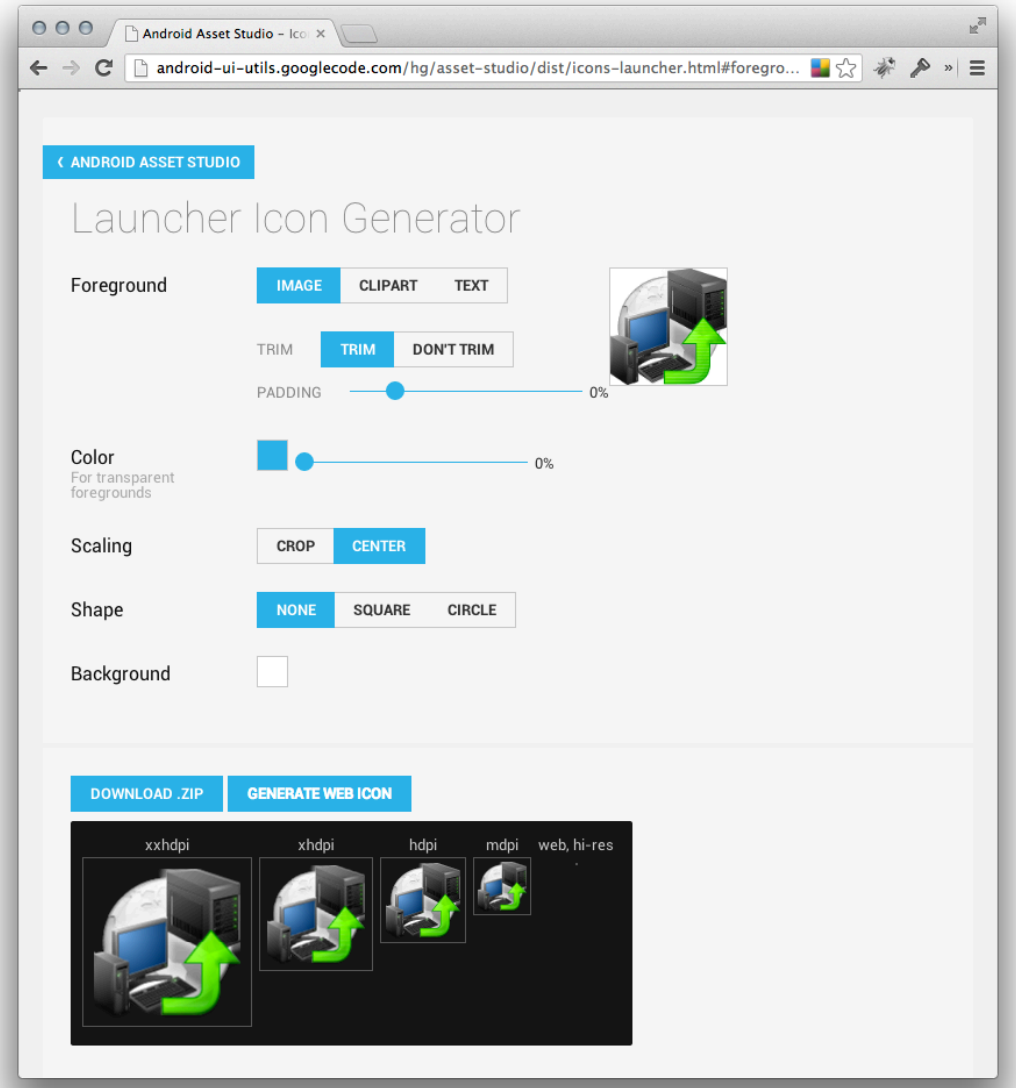
For guidance on supporting different device factors, refer to the Android Resources guide.

Creating Resources with Android Asset Studio

Creating resources for all the possible screen sizes and densities can be tedious. To help streamline this process, Google created an online utility called the Android Asset Studio that can reduce some of the tedium involved with the creation of these images:



By uploading an image, Android Asset Studio can then create bitmaps that will target the most common screen densities. Android Asset Studio will then allow the bitmaps to be downloaded as a zip file:



The Android Asset Studio is a highly recommended tool and will simplify this process greatly.

Summary

In this chapter, we examined how to use Xamarin.Android to create and deploy an Android application. We looked at the various parts of an Android application that was created by using the default Xamarin.Android application template. We then wrote an application of our own. We first created some string resources, and then used the Xamarin Designer to create the user interface. We wrote some code to translate the Phoneword into a phone number and placed a phone call. In order to make that happen, we also learned about Intents, and saw how to add

permissions to our application so that Android would allow the phone call to be placed.