Hello, Android

Creating your first Xamarin Android app

Overview

In this tutorial we'll look at how to create, deploy, and run a Xamarin.Android application. First, we'll demonstrate how to use the default application template in the deployment process. Next, we'll examine some of the basic parts of the android application that are created with the template. We'll then create a hello world application, showing how to build the user interface both in code and by using Android XML.

Creating a Xamarin.Android Application

To get started, we are going to walk through the steps you need to take to create a Xamarin.Android application and deploy it to the emulator. Xamarin.Android works with Xamarin Studio on both OSX and Windows; it also works on Windows with Visual Studio 2015 Community (or greater). The process for creating Xamarin.Android applications is nearly the same on each of these platforms. This walkthrough assumes you already have Xamarin.Android installed and that you have created an emulator. If that's not the case, refer to the Installation document before continuing.

Creating a New Application

Let's begin by creating a new Xamarin.Android solution. Xamarin.Android includes several templates for creating projects, including:

- Blank App – A basic starter project with a single activity.
- OpenGL Game – An OpenGL starter project.
- Class Library– A reusable .NET library project for Android.

We're going to use the Blank App template for this walkthrough. Let's create an application by doing the following:

New Solution-- Visual Studio

1. From the file menu, select New > Project ...
2. In the New Project dialog select Android > Blank App (Android)
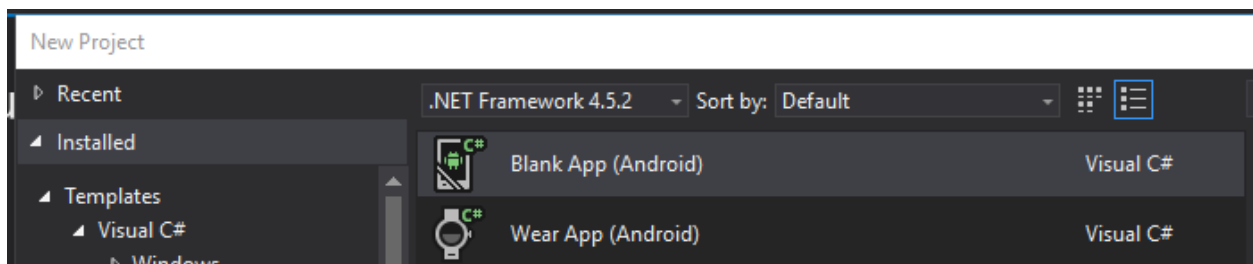
3. Enter the name: "HelloAndroid", and select an appropriate location for the solution folder, click OK.
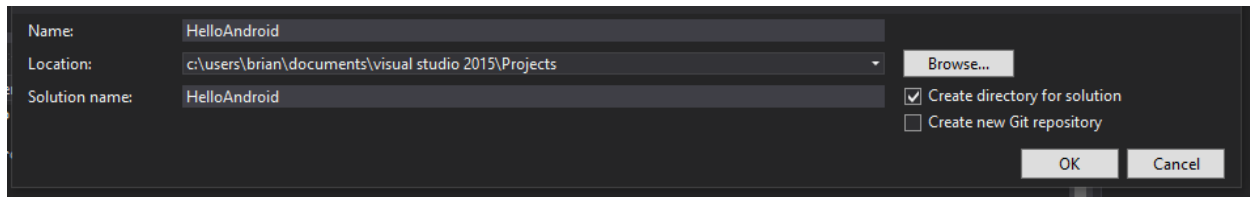
4. In the Solution Explorer, rename the project to HelloAndroid.Code

Solution Components

In the previous section, we created a simple Xamarin.Android project. In doing so, the Xamarin.Android template generated a variety of things for us. Let's take a look at what it created.

The HelloAndroid project includes four folders named Properties, References, Assets, and Resources. These items are summarized in the table below:

Folder	Purpose
Properties	Contains normal .NET assembly metadata.
References	Packages and other projects that this project uses (dependencies).
Assets	Contains any type of file the application needs included in its package. Files included here are accessible at runtime via the Assets class.
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.

Android Activities

The Xamarin.Android template also created a file named MainActivity.cs which contains a class that inherits from Activity. 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. An Activity contains methods to be called at certain points in the lifecycle. These methods can also be overridden. For example, 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 after you finish the Getting Started series of tutorials, we recommend reading the Activity Lifecycle tutorial.

Creating the User Interface with C# Code

Now let's create the user interface in code. Earlier, we talked briefly about Activities and mentioned that activities have a lifecycle. After an Activity starts, its OnCreate method is called. This is the appropriate place to perform initialization of the application, such as loading the user interface for the Activity. For

our app, we need to create a Button and a TextView. When the user clicks the button, we want the TextView to display a message. To accomplish this, open the MainActivity.cs file and replace the OnCreate method with the following code:

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

    //Create the user interface in code
    var layout = new LinearLayout(this);
    layout.Orientation = Orientation.Vertical;

    var aLabel = new TextView(this);
    aLabel.Text = "Hello, Xamarin.Android";

    var aButton = new Button(this);
    aButton.Text = "Say Hello";
    aButton.Click += (sender, e) => {
        aLabel.Text = "Hello from the button";
    };
    layout.AddView(aLabel);
    layout.AddView(aButton);
    SetContentView(layout);
}
```

Now that we've implemented our own simple Hello, World app, let's run the
This will let test the app and it will help you become familiar with the process of deploying and launching an app in the emulator.

Launching the Emulator – Visual Studio

We can run the app as usual in Visual Studio by choosing Debug > Start Without Debugging (or Start Debugging to run with the debugger attached), or by clicking on the green run button. But first you should select an emulator image from the drop-down list to the right of the run button.

Deploying and Launching the Application

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. After the emulator starts up, slide the lock button to the right to show the Android home screen:

Back in the Select Device dialog, we can now build and deploy our app by selecting the emulator in the list, and then clicking OK. 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. Xamarin.Android deploys the app to the emulator, and then launches it. The default app will display a label and a Button. as shown below:
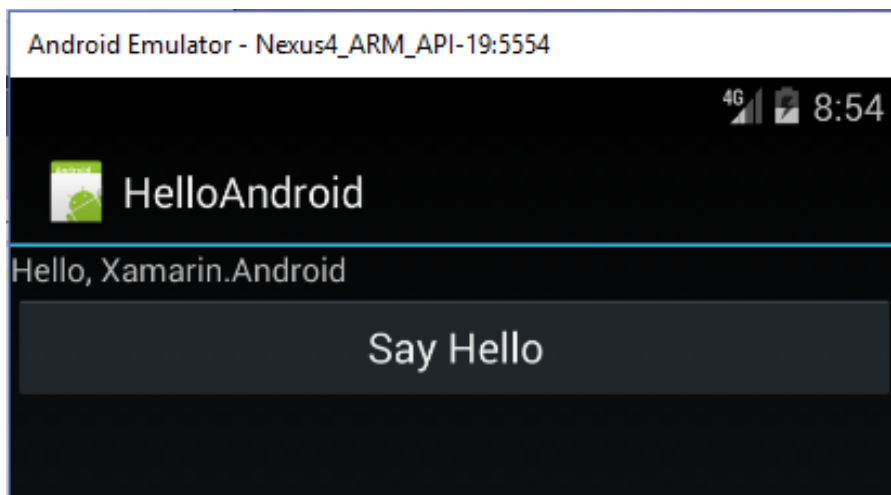
When you click on the "Say Hello" button, the message displayed in the label will change as shown below:
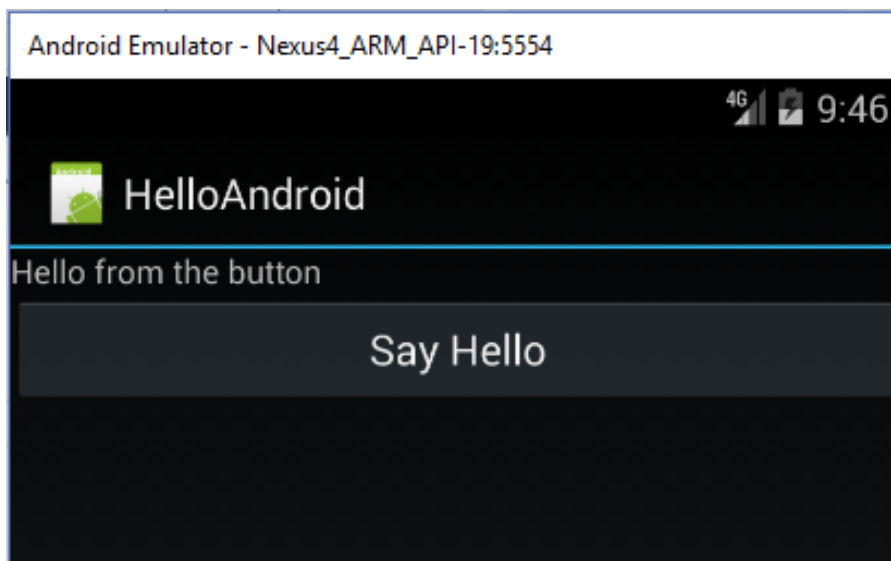
Code Walkthrough

Let's break this code down line-by-line. First we created a LinearLayout and set its Orientation to Vertical with these lines:

```
var layout = new LinearLayout(this);
layout.Orientation = Orientation.Vertical;
```

Android uses layout classes to group and position controls on the screen. The controls we add, such as the Button and the TextView, will be children of the layout. The LinearLayout class is used to align controls one after another, either horizontally, or vertically as we have done here.

Next, we created a Button and TextView, setting the Text property of each like this:

```
var aLabel = new TextView(this);
aLabel.Text = "Hello, Xamarin.Android";

var aButton = new Button(this);
aButton.Text = "Say Hello";
```

When the user clicks the button, we want to change the text of the TextView. With Xamarin.Android, we accomplish this by using a typical .NET event. To handle such an event, we can use an event handler, an anonymous method, or even a lambda expression as in the following example:

```
aButton.Click += (sender, e) => {
    aLabel.Text = "Hello from the button";
};
```

Instead, we could use a C# anonymous method with the delegate syntax:

```
aButton.Click += delegate(object sender, EventArgs e) {
    aLabel.Text = "Hello from the button";
};
```

With the controls created and the event handler wired up, we need to add them to the LinearLayout instance. LinearLayout is a subclass of ViewGroup. A ViewGroup is basically a view that contains other views and determines how to display them. The ViewGroup class contains an AddView method that we can call to add our controls, as we did in this code:

```
layout.AddView(aLabel);
layout.AddView(aButton);
```

The final step in building our app is to add the layout to the screen. We accomplished this by calling the Activity's SetContentView method, passing it the layout like this:

```
SetContentView(layout);
```

Creating this code shows how easy and familiar programming applications in Xamarin.Android can be for a .NET developer. However, something unique to Android is how it manages resources. In our simple

example, we hard-coded all the strings. Let's change this to use the Android resource system and show you how to make your programming process even simpler and more foolproof.

Creating String Resources

As we just mentioned, Android accesses resources in an unusual way. Android resources are managed under the Resources folder in the Solution Explorer. To make them accessible from code, Xamarin generates Resource classes which are updated for all resources that are included in the various Resources subfolders.

Let's look at an example where we replace our hard-coded strings with string resources. Under the Resources > values folder, open the file named Strings.xml. This file includes the application's string resources. Replace the contents of the file with the following code:

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">HelloAndroid</string>
    <string name="helloButtonText">Say Hello</string>
    <string name="helloLabelText">Hello Xamarin.Android</string>
</resources>
```

We defined two string resources in the XML above, one with the name helloButtonText and another named helloLabelText. Each of these contains a string value. When we include values like these in the Strings.xml file, the generated Resource class will be updated when we rebuild, giving us a mechanism to access the resources from code.

The auto-generated Resource class is in the file Resource.designer.cs. As it is auto-generated, it should never be changed by hand. For the strings we just added, a nested class named Strings will be created in the Resource class. The integer fields in this class identify each string, as shown in the following code:

```
public partial class String {
    // aapt resource value: 0x7f040000
    public const int helloButtonText = 2130968576;

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

    private String(){}
}
```

To access the strings from code, we call the SetText method of the TextView and Button controls respectively, passing the appropriate resource id. For example, to set the TextView's text, replace the line where we set the aLabel.Text property with a call to the SetText method like this:

```
aLabel.SetText(Resource.String.helloLabelText);
```

Likewise, to set the Button's text, we call its SetText method:

```
aButton.SetText(Resource.String.helloButtonText);
```

When we run the application, the behavior is the same using either technique, only now we can easily manage the string values without needing to change them in code.

Resources also come into play when we use XML to declare our user interface, as we'll see in the next section.

Creating the User Interface with XML

In addition to creating the user interface in code, Android supports a declarative, XML-based user interface system, similar to technologies such as XAML and HTML. Let's modify our example to create the UI with XML.

Creating a Second Project

We will add a second project to our solution that has the same functionality as the first, but accomplishes this using a UI defined in XML.

1. In the Visual Studio Solution Explorer, right-click on the solution and select: Add > New Project …
2. Select Blank App (Android) and name the project HelloAndroid.Xml
3. Copy the code in MainActivity.OnCreate from the first project and paste it into MainActivity.OnCreate in the new one.
4. Copy the string definitions from Strings.xml in the first project to Strings.xml in the new one.

Under the Resources > layout folder is a file called activity_main.axml. Let's change this file to contain a LinearLayout with a TextView and a Button, just as we've done earlier when we used code to create this application. We can accomplish this by changing activity_main.axml to include the following 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:id="@+id/helloLabel"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/helloLabelText" />
 <Button
        android:id="@+id/aButton"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/helloButtonText" />
 </LinearLayout>
```

If we include XML elements like these for the LinearLayout, TextView, and Button classes, it will cause instances of these to be created at runtime. To access these instances from code, we use their resource ids.

Resource IDs

The syntax @+id/name tells the Android parser to generate a resource id for the given element with the supplied name. For example, when we give the TextView class an id of @+id/helloLabel, the Resource class will have a nested class Id with the integer field helloLabel. Similarly, the Id class will also contain a field named aButton, as shown in the code below:

```
public partial class Id {
      // aapt resource value: 0x7f050001
      public const int aButton = 2131034113;

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

      private Id()
      {
      }
}
```

Also, the syntax @string/name allows us to access the string resources that we created earlier in Strings.xml.

Just like the string resources we saw earlier, we can access these controls by using the generated Resource class, only this time through the Id subclass. Let's switch back over to the Activity1.cs file to show how.

In the OnCreate method, we now only need code to set the content view and create the event handler, since the LinearLayout, TextView, and Button classes are created from the XML we just defined. Change the OnCreate method with the following code:

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

   SetContentView(Resource.Layout.Main);

   var aButton = FindViewById<Button> (Resource.Id.aButton);
   var aLabel = FindViewById<TextView> (Resource.Id.helloLabel);

   aButton.Click += (sender, e) => {
       aLabel.Text = "Hello from the button";
   };
}
```

We still needed to call SetContentView as we did in our earlier example, only this time we passed the resource id for the layout, Resource.Layout.Main, which we defined in activity_main.axml.

Next, we simply looked up the Button and TextView instances, respectively, by using the FindViewById method and passing the appropriate resource ids. Finally, we set the event handler, as we did earlier in the Creating the User Interface with Code section.

When you run the app now, you'll see that it works exactly as it did in the earlier implementation where everything was accomplished in code.

Summary

In this article, 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 walked through how to use only code to create a simple application. Finally, we used Android XML and a declarative user interface definition to create the same application we'd just created in code.

Attribution

This tutorial was adapted and updated April 2, 2017 and January 12, 2018 by Brian Bird from a 2012 Xamarin tutorial titled "Hello, Android". That tutorial is no longer available on the Xamarin web site, but in 2012 it was located at: http://docs.xamarin.com/guides/android/getting_started/hello%2C_world