An Introduction to Xamarin: Part 1

by Derek Jensen 6 Jun 2014
Difficulty: Beginner Length: Long Languages: English ▼

Mobile Development Xamarin Android SDK

This post is part of a series called An Introduction to Xamarin.
An Introduction to Xamarin: Part 2

From Humble Beginnings

It's amazing to think that almost ten years ago, when Mono was officially released, C# developers would have the vast majority of the mobile landscape at their fingertips. I remember exactly where I was. It was the summer of 2004 and I was putting the finishing touches on a fat-client desktop application using the .NET Framework 2.0.

I was creating an application for visualizing corporate data centers in Visio and automatically generating migration plans and checkpoints for virtualizing their environments. It was groundbreaking stuff at the time if you ask me.

I was trying to stay on top of .NET as best I could, so when I heard there was going to be an open version, I thought, "neat". Maybe I could run my application on a Linux machine. But I was working at a Microsoft shop and didn't see much use in it, so I dismissed it for a while.

About a year before Mono went live, the company that created it, Ximian, was purchased by Novell, and work on its products continued. Among these products was Mono. In its time under the umbrella of Novell, Mono continued to be improved closely following the growth and functionality of the .NET Framework through Microsoft.

During this time, two very large advancements in the mobile space regarding Mono arrived, **MonoTouch** and **Mono for Android** were released in 2009 and 2011, respectively. Much to the amazement of the .NET community, we could now write mobile apps that targeted the iOS and Android platforms in a language that we were familiar with. Unfortunately, this wasn't immediately met with open arms.

While the Android platform didn't seem to have much trouble with this, Apple on the other hand, wasn't quite as receptive. In mid-2010, Apple updated the terms of their iOS Developer Program that prohibited developers from writing apps in languages other than C, C++, and Objective-C, and restricted any sort of layer between the iOS platform and iOS applications.

This could certainly have spelled disaster for MonoTouch going forward. Luckily, in late 2010, Apple relaxed the language restrictions and the future of MonoTouch looked bright again, even if only briefly.

As the outlook for MonoTouch users began to look bright again, there was another snag. In early 2011, Attachmate acquired Novell and announced hundreds of layoffs of the Novell workforce. Among those layoffs were several of the founders of the original Mono framework as well as the architects and developers of both MonoTouch and Mono for Android. Once again, we became concerned about the future of our ability to create C# apps running on these new platforms.

Hardly a month after being laid off, Miguel de Icaza created a new company named Xamarin and vowed to continue the development and support of Mono. Novell and Xamarin announced that a perpetual license of Mono, MonoTouch, and Mono for Android would be granted and that Xamarin would now officially take over the project. We once again had the keys to the kingdom.

Getting Up and Running

Roughly three years after the creation of Xamarin, we are left with some truly remarkable tools. These tools are not only remarkable for the fact that they allow us to write C# applications that run on non-Microsoft platforms, but they also are extremely easy to get up and running.

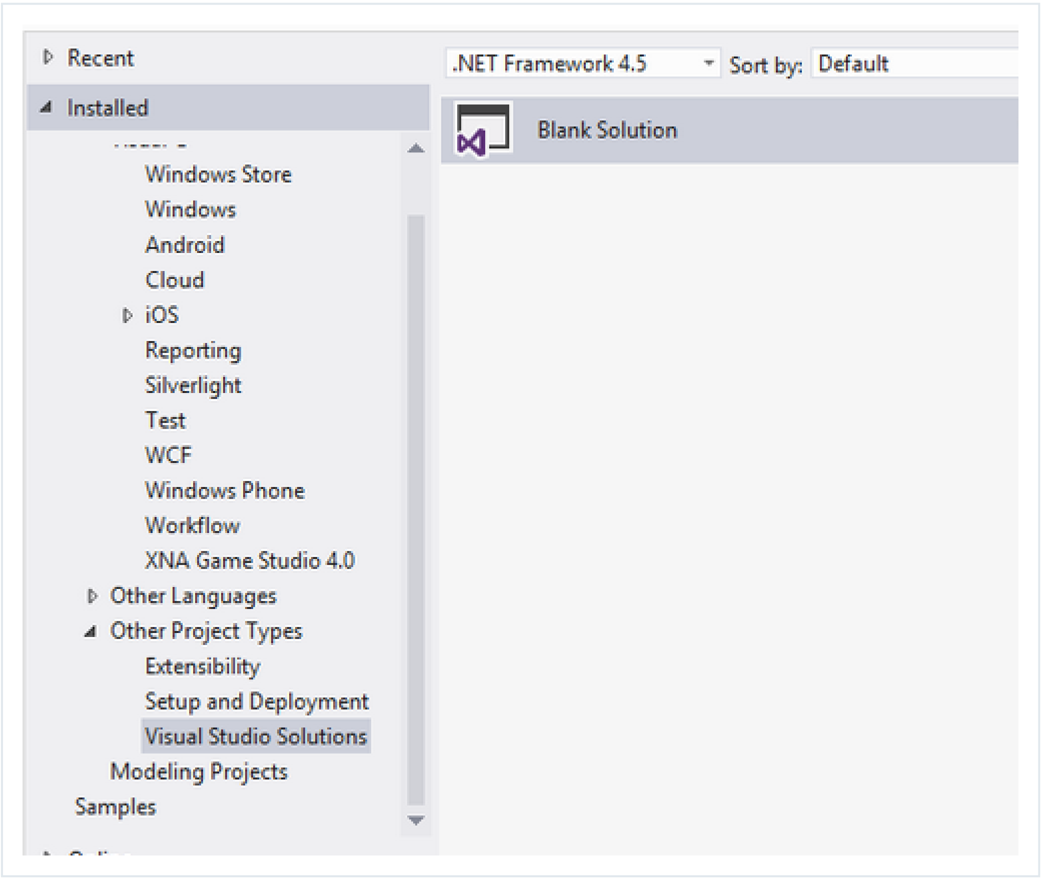
Step 1: Installation

To get started, you simply head over to the Xamarin website, sign up for an account if you don't already have one, and visit the download page. With every account, you get a free 30-day trial of the Business Edition of Xamarin, which provides you with everything you need.

In the last several years, the installation process of Xamarin has improved greatly from the days of MonoTouch and Mono for Android. It's a completely self-contained installation that will detect the required software and their versions to get started, including the appropriate version of the Android SDK.

Step 2: Development Environments

In my mind, the most important feature of the Business (and Enterprise) Edition is its support for Visual Studio. This means you can write all of your iOS and Android applications using not only an IDE that you're comfortable with, but also get the added benefit of any other plugins or extensions for Visual Studio you may be using, Resharper for example.

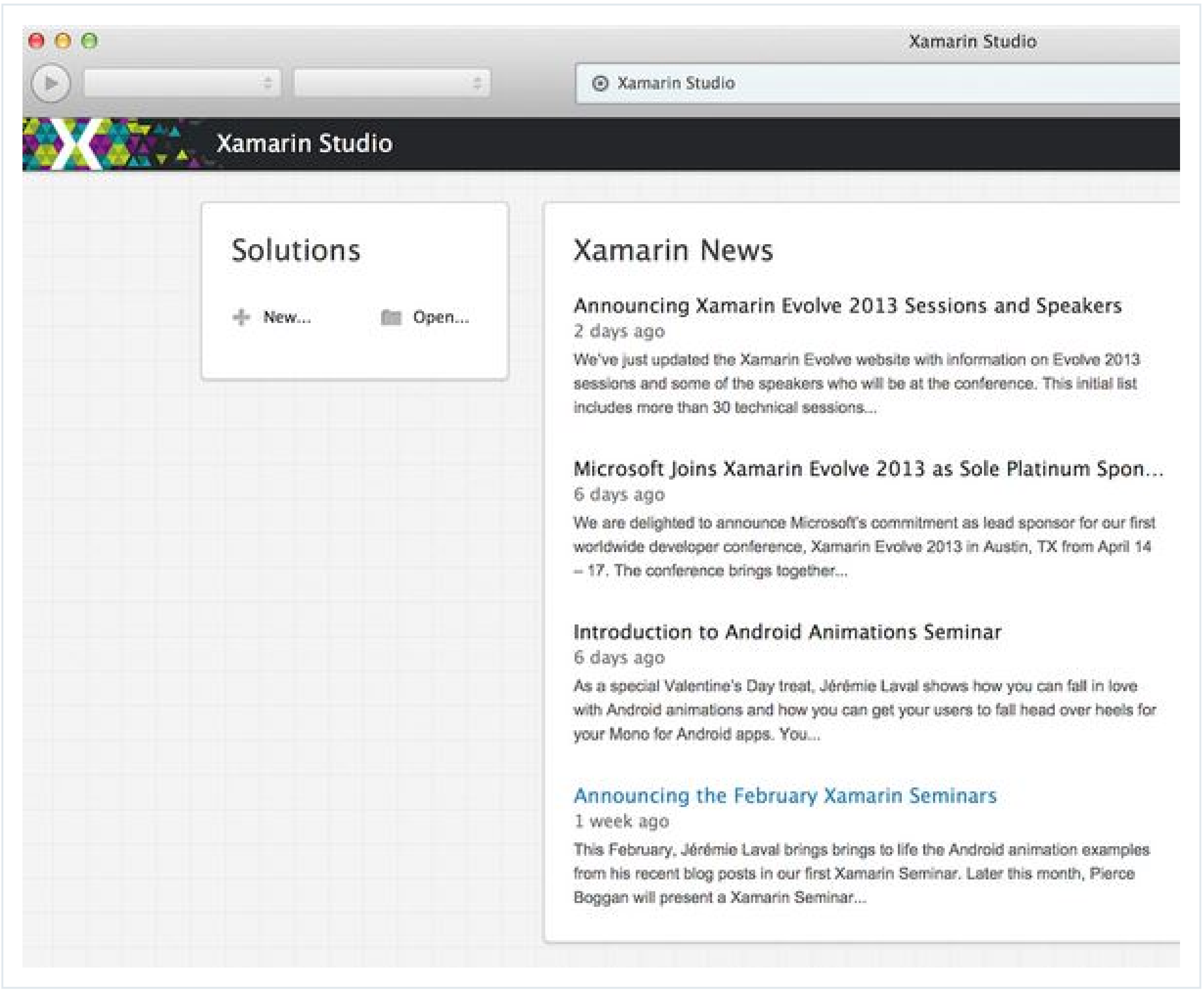


I don't know about you, but I sure get a jolt of excitement when I open up Visual Studio, select **File > New Project** and stare straight in the face of the options to create an iOS or Android application.

If your 30-day free trial of the Business Edition has expired by the time you read this, you can simply downgrade to the Starter Edition and continue to play with Xamarin. However, there are a couple of drawbacks to the Starter Edition.

- You're no longer able to use Visual Studio.
- Your application has a size restriction.

If you're simply using the Starter Edition to play around with Xamarin, these restrictions are no big deal. If you're working on the next big app, though, you will need to pony up for the Business or Enterprise Edition. Every edition also comes with a free IDE, Xamarin Studio.



Xamarin News

Announcing Xamarin Evolve 2013 Sessions and Speakers
2 days ago
We've just updated the Xamarin Evolve website with information on Evolve 2013 sessions and some of the speakers who will be at the conference. This initial list includes more than 30 technical sessions...

Microsoft Joins Xamarin Evolve 2013 as Sole Platinum Spon...
6 days ago
We are delighted to announce Microsoft's commitment as lead sponsor for our first worldwide developer conference, Xamarin Evolve 2013 in Austin, TX from April 14 – 17. The conference brings together...

Introduction to Android Animations Seminar
6 days ago
As a special Valentine's Day treat, Jérémie Laval shows how you can fall in love with Android animations and how you can get your users to fall head over heels for your Mono for Android apps. You...

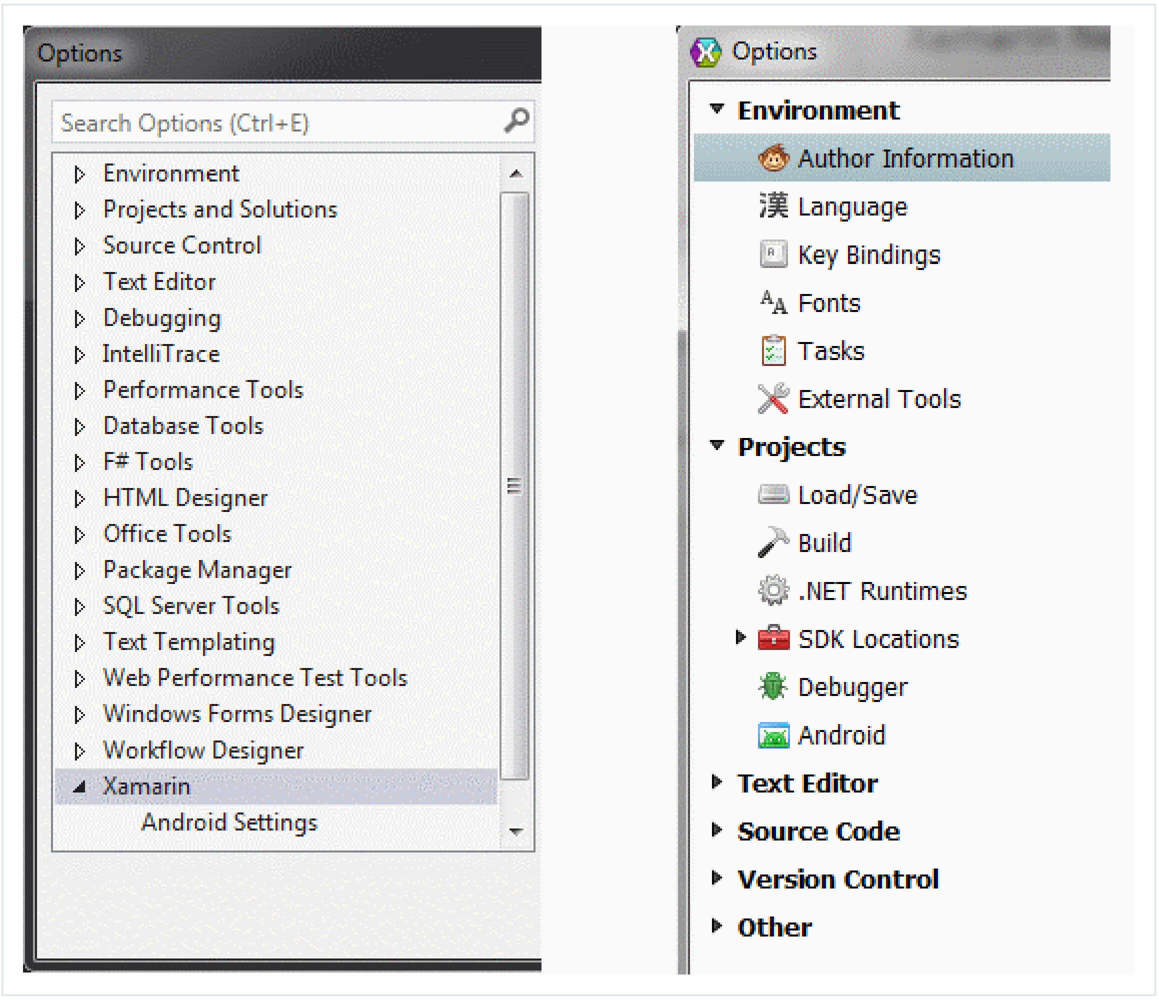
Announcing the February Xamarin Seminars
1 week ago
This February, Jérémie Laval brings brings to life the Android animation examples from his recent blog posts in our first Xamarin Seminar. Later this month, Pierce Boggan will present a Xamarin Seminar...

Xamarin Studio is a full-featured IDE that includes many features you also find in Visual Studio, so you definitely don't need to feel shorted in any way if you choose to use Xamarin Studio. I feel very comfortable using it and it's truly a joy to work with.

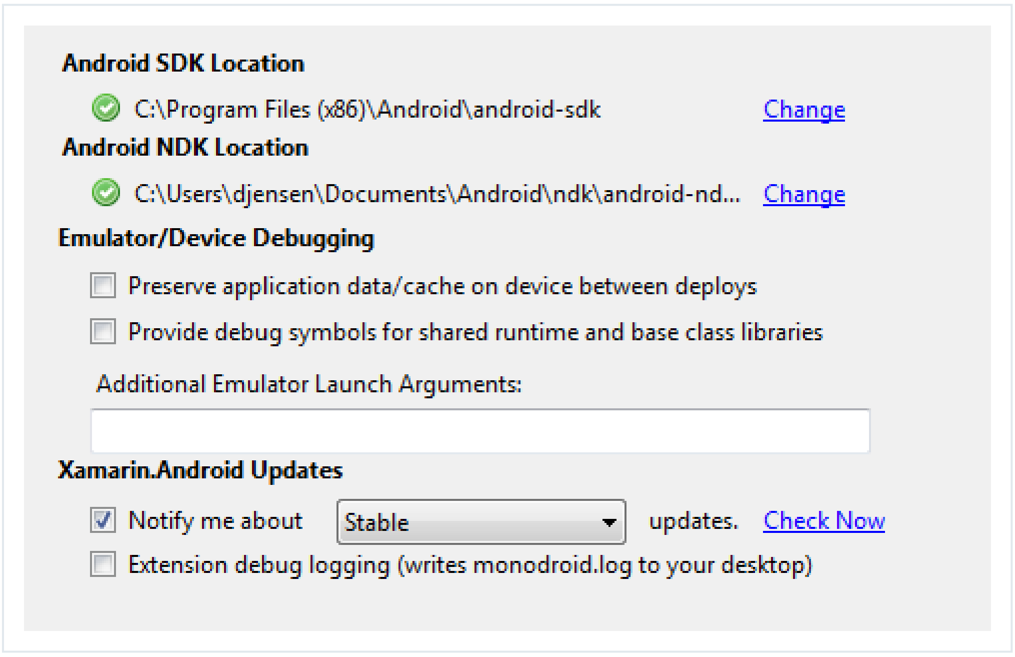
The nice thing is that the solution and project structures are interchangeable with those of Visual Studio. This means that if you have a license for an edition of Xamarin that allows you to use Visual Studio, you can work on the same solution in either IDE. This enables cross-team development between developers that use either a Windows or Mac based system. There is no need for virtualization software, because Xamarin Studio is available for both Windows and OS X.

Step 3: Configuration

When you get started with Xamarin, it's important to be aware of the configuration options. To get into the basic configuration, select **Tools > Options** from Visual Studio or Xamarin Studio.



I currently only have **Xamarin.Android** installed. If you've also installed **Xamarin.iOS**, you will see more configuration options. From the right-side of the options dialog you will see the following options in Visual Studio.


Android SDK Location

✓ C:\Program Files (x86)\Android\android-sdk — Change

Android NDK Location

✓ C:\Users\djensen\Documents\Android\ndk\android-nd... — Change

Emulator/Device Debugging

☐ Preserve application data/cache on device between deploys

☐ Provide debug symbols for shared runtime and base class libraries

Additional Emulator Launch Arguments:

Xamarin.Android Updates

☑ Notify me about Stable updates. Check Now

☐ Extension debug logging (writes monodroid.log to your desktop)

In Xamarin Studio, similar options are split across the **SDK Locations**, **Debugger**, and **Android** tree view items in the **Projects** section. Let me walk you through the various configuration options.

Android SDK and NDK Locations

As you may have guessed, these settings are used to set the location of the Android bits on your machine. I typically don't find it necessary to modify these and most often a clean installation of Xamarin—sometimes with an update or two—will download and install all the appropriate versions in the correct locations. If you're a more seasoned Android developer who needs to have access to multiple versions and be able to switch back and forth, you have that ability.

Preserve application data/cache on device between deploys

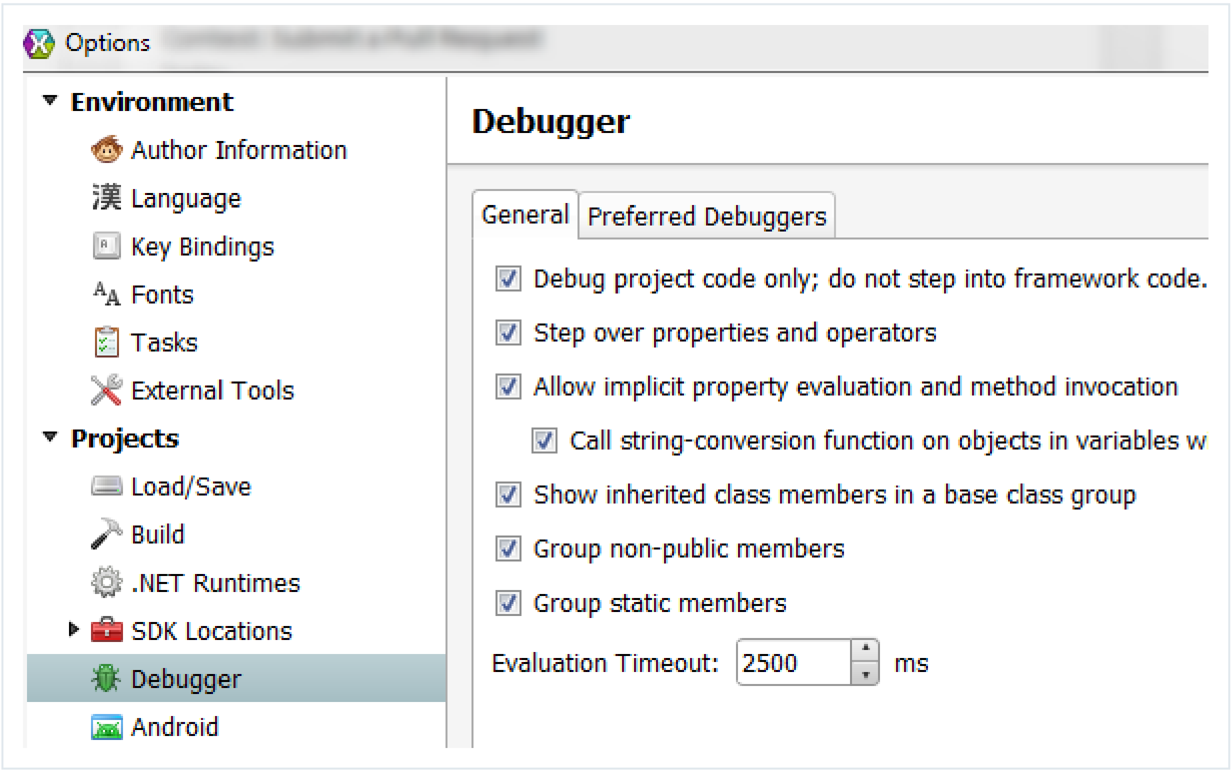
This is probably the configuration option that I use most. When I'm writing an application that works with local sandbox data on a device or in the emulator, such as files or a database, I will eventually check this box.

During the deployment of an application to a device or the emulator, all existing data-including database files—are removed and must be created again. In the early stages of development, when I want to make sure the database is being created successfully, this is fine. After that point, however, I'll want to work with a populated database and not have to set that data up every time the application is deployed.

Provide debug symbols for shared runtime and base class libraries (Visual Studio only)

During the development cycle, you're deploying your debug build to a device or on the emulator. By default, bundled with your application are your debug symbols that allow you to debug, set breakpoints in your own code, and step through lines while the application is running. This option allows you to also have access to the debug symbols in the shared Mono runtime as well as the base class libraries to give you more information about what's happening in those areas as well as your own code.

This is only used for debugging purposes. A similar option is found in Xamarin Studio. You can find it under the **Debugger** option as **Debug project code only; do not step into framework code.** You will need to uncheck this checkbox to step into the Mono framework.



Debugger

- [✓] Debug project code only; do not step into framework code.
- [✓] Step over properties and operators
- [✓] Allow implicit property evaluation and method invocation
  - [✓] Call string-conversion function on objects in variables w
- [✓] Show inherited class members in a base class group
- [✓] Group non-public members
- [✓] Group static members

Evaluation Timeout: 2500 ms

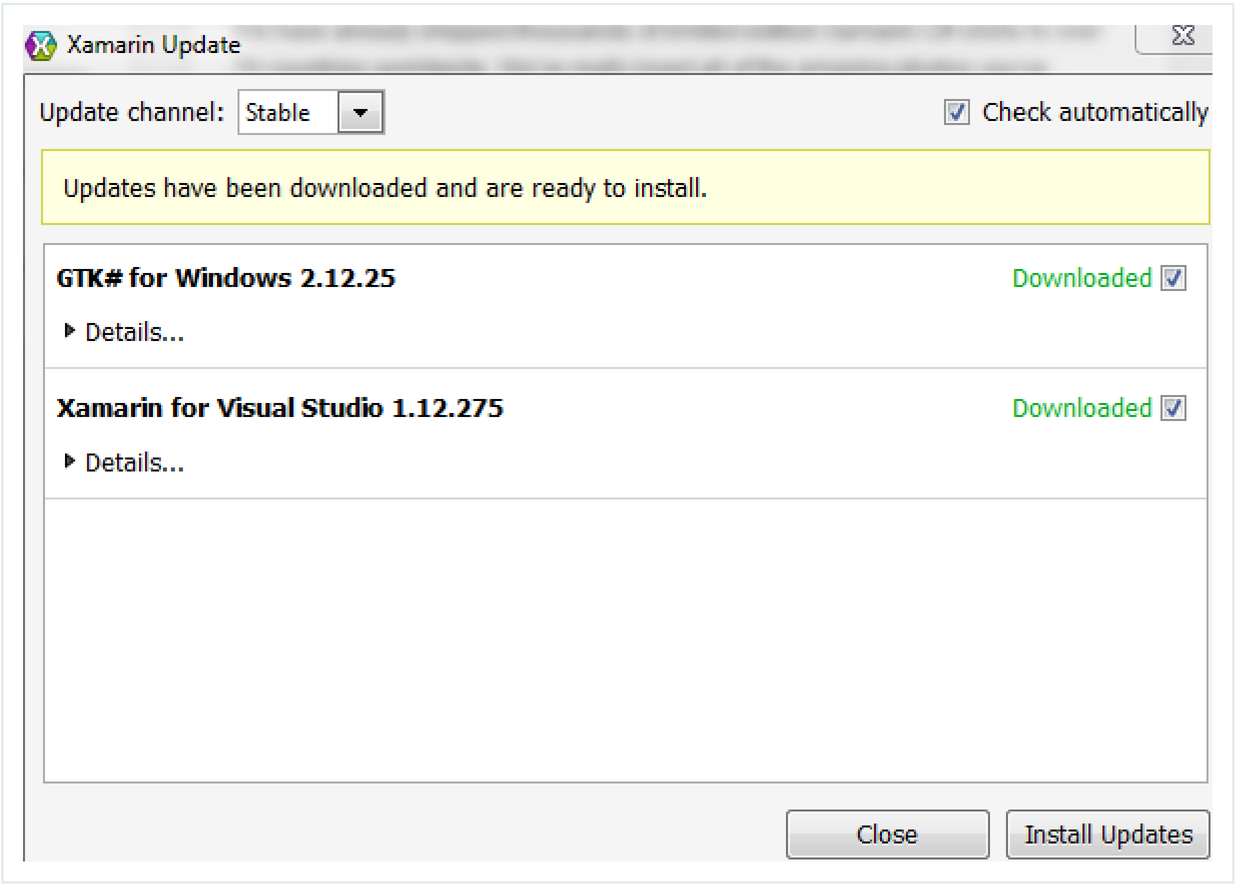
Additional Emulator Launch Arguments

If you have the need to tweak the Android emulator with additional settings that you could typically set manually when running, this option allows you to pass those arguments directly to the emulator through Visual Studio or Xamarin Studio.

Notify me about updates (Visual Studio only)

The Xamarin software is constantly evolving and being updated. It really pays to stay on top of any changes to the version you're currently using as well as what's coming. Here is where you can set which types of updates you want to be notified of. You will be asked to download and install new versions if you check this checkbox.

I typically stay with stable releases for applications that are scheduled for release, but I like to have an alpha or beta to play with. There's a similar option in Xamarin Studio, but in a different location. It can be found under **Help > Check for Updates**. Here you can choose the **Update channel** as **Stable**, **Beta**, or **Alpha**, just as in Visual Studio.



Update channel: Stable
[✓] Check automatically

Updates have been downloaded and are ready to install.

GTK# for Windows 2.12.25 — Downloaded [✓]
Details...

Xamarin for Visual Studio 1.12.275 — Downloaded [✓]
Details...

Extension debug logging (writes monodroid.log to your desktop) (Visual Studio only)

This option enables device deployment logging. When this option is enabled, Visual Studio directs output from the deployment log to the **monodroid.log** file found on your desktop. This option is not available in Xamarin Studio, at least not as a configuration.

Xamarin Studio always writes device deployment logs, but they are a little bit more difficult to find. On Windows, you can find them in the `\LOCALAPPDATA\XamarinStudio-{VERSION}\Logs` folder where `VERSION` is the version of Xamarin Studio you're using. The files are created in that folder with the naming convention of `AndroidTools-{DATA}__{TIME}` where `DATE` is the current date of the deployment and `TIME` is the actual time of the deployment.

Writing Code

Before I address the beauty of this code, and boy is it beautiful, it is important for you to understand that just because you can write iOS and Android applications in C#, doesn't mean you can just write iOS and Android applications in C#. The Xamarin team has done a fantastic job enabling developers with a background in C# to have the *ability* to create iOS and Android applications. The problem lies in knowledge. Let me explain this in more detail.

You may have the C# knowledge you need, but unless you have dabbled in iOS or Android development in the past, you don't have the platform knowledge. In order to make Xamarin usable to all levels of iOS and Android developers, the Xamarin team has mapped the language constructs and class structures from Objective-C for iOS and Java for Android into C#.

So what does that mean? It means that you at least need to have a basic understanding of the iOS and Android programming model and SDKs to be able to take advantage of Xamarin. That means that the `AVFoundation` class in Objective-C is the `MonoTouch.AVFoundation` class in C#. The `Activity` class in Java is the `Android.App.Activity` class in C#.

If you don't have any experience with iOS or Android, then don't let your lack of knowledge deter you from using Xamarin. You don't need to spend months, days, or even hours on the iOS Dev Center or the Android Developer website. The point is that becoming familiar with the platform you're developing for is more than worth your time if your ambition is to create a high quality product.

My suggestion is to go straight to the Xamarin Developer Center and get up and running quickly. You will find documentation, sample applications, tutorials, videos, and API references. Everything after getting started is simply researching how to accomplish certain tasks. Once you get a good handle on the APIs and the development flow, you can go back to the iOS and Android resources to get a more in depth knowledge of the platforms.

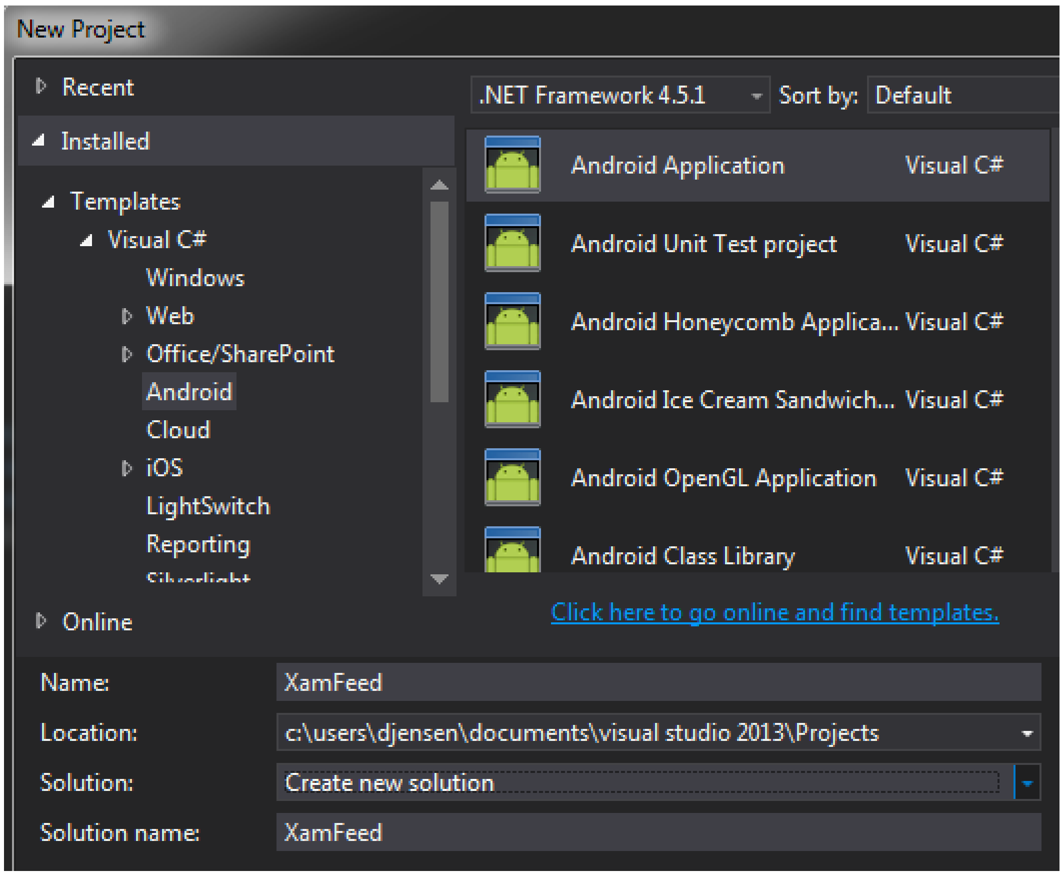
Let's Create an Application

Now that you have the necessary tools downloaded and installed, let's take them for a spin. To follow along, you can use either Visual Studio or Xamarin Studio, because I'll be focusing on the code, not the IDE. For this tutorial, I'll be using Visual Studio 2013 running on Windows, but you're free to use either IDE on Windows or Xamarin Studio on OS X. We will be creating a simple Android application that will read the current news feed from Xamarin and we'll call it **XamFeed**.

Create A Xamarin.Android Project

Start as you would with any other C# application by creating a new project/solution and naming it **XamFeed.** Typical naming conventions for an application like this would append **.Android** to the end of the name. This is to differentiate the name of this application from any other platform specific version you may create later (like **.iOS**, **.Mac**, **.WindowsPhone**, etc).

This will be a very simple application, so we will keep the name simple as well. You can choose any of the Android templates you want, Android Application, Android Honeycomb Application, or Android Ice Cream Sandwich Application. These just set the base version of Android that our application will target. I will use the basic **Android Application** template.



Android Application	Visual C#
Android Unit Test project	Visual C#
Android Honeycomb Applica...	Visual C#
Android Ice Cream Sandwich...	Visual C#
Android OpenGL Application	Visual C#
Android Class Library	Visual C#

Name: XamFeed
Location: c:\users\djensen\documents\visual studio 2013\Projects
Solution: Create new solution
Solution name: XamFeed

Write Some Code

In the **Solution Explorer**, open the `MainActivity` class, which will be the main entry point of our application. I like to rename this `Activity` to better represent the purpose it will serve, so go ahead and rename it to `FeedActivity`.

If you're unfamiliar with activities, think of an `Activity` as a screen or view within your Android application. Each screen you need in your application will have a corresponding class that inherits from the `Activity` base class.

In the `FeedActivity` class, you have the ability to override a number of methods that are provided for you out of the box. The only one that we are concerned about at the moment is the `OnCreate` method that will be called when our screen is created and accessible to the user.

The first thing we'll do is create a new class that represents the feed. You can obviously expand upon this, but all we need for now is the `Title`, `PubDate`, `Creator`, and `Link` to the item's content.

```
public class RssItem
{
    public string Title { get; set; }
    public string PubDate { get; set; }
    public string Creator { get; set; }
    public string Link { get; set; }
}
```

We can now change the implementation of the `OnCreate` method within our `MainActivity` class to get the data from the Xamarin feed. Replace the `OnCreate` implementation with the following:

```
[Activity(Label = "XamFeed", MainLauncher = true, Icon = "(
public class FeedActivity : ListActivity
{
   private RssItem[] _items;

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

      using (var client = new HttpClient())
      {
         var xmlFeed = await client.GetStringAsync("http:/,
         var doc = XDocument.Parse(xmlFeed);
         XNamespace dc = "http://purl.org/dc/elements/1.1/'

         _items = (from item in doc.Descendants("item")
                   select new RssItem
                   {
                       Title = item.Element("title").Value,
                       PubDate = item.Element("pubDate").V
                       Creator = item.Element(dc + "creato
                       Link = item.Element("link").Value
                   }).ToArray();

         ListAdapter = new FeedAdapter(this, _items);
       }
    }

    protected override void OnListItemClick(ListView l, Vi
    {
       base.OnListItemClick(l, v, position, id);

       var second = new Intent(this, typeof(WebActivity));
       second.PutExtra("link", _items[position].Link);
       StartActivity(second);
    }
}
```

Let's walk through this code snippet line by line to see what's going on.

```
[Activity(Label = "XamFeed", MainLauncher = true, Icon = "@
```

The `ActivityAttribute` that decorates the `FeedActivity` class is the mechanism that **Xamarin.Android** uses to let the target device or the emulator know that this is an `Activity` (or screen) that is accessible within the application. This is required for all `Activity` classes within your application.

```
private RssItem[] _items;
```

We are going to save all the feed items that we pull from the Xamarin website in a variable to prevent us from constantly making HTTP requests. You may want to handle this differently depending on whether or not you want to update this screen later with new content. In our simple application, we won't do this.

```
protected async override void OnCreate(Bundle bundle)
```

Next, we override the `OnCreate` method that's exposed through the `Activity` base class in our `FeedActivity` class. This method is called every time this `Activity` is instantiated by the. As you can see, we can also use the new C# 5.0 `async/await` feature to make this method asynchronous.

```
base.OnCreate(bundle);
```

Make sure to call the `base.OnCreate` method on the base `Activity` class. This will ensure that any processing the base class does during the `OnCreate` method will continue to run.

```
using (var client = new HttpClient())
```

To fetch the RSS data from the Xamarin website, we're going to use the `HttpClient` class as it provides a number of convenient asynchronous methods to retrieve data over HTTP.

```
var xmlFeed = await client.GetStringAsync("http://blog.xamar
var doc = XDocument.Parse(xmlFeed);
```

We then invoke the `GetStringAsync` method on the `HttpClient` class to retrieve the feed data and `Parse` it into an `XDocument` object to do some **Linq2XML** magic.

```
XNamespace dc = "http://purl.org/dc/elements/1.1/";
_items = (from item in doc.Descendants("item")
    select new RssItem
    {
        Title = item.Element("title").Value,
        PubDate = item.Element("pubDate").Value,
        Creator = item.Element(dc + "creator").Value,
        Link = item.Element("link").Value
    }).ToArray();
```

To correctly retrieve elements from the resulting `XDocument` object, we need to create an instance of the `XNamespace` class that represents some of the custom formatting used within the Xamarin RSS feed. We can then run a LINQ query against the `XDocument` to pull all the items out and create new instances of the `RssItem` class based on the item properties.

```
ListAdapter = new FeedAdapter(this, _items);
```

Finally, we use a custom adapter to populate the `ListView` of the `FeedActivity`, which is defined in the `Main.axml` document in the `Resources/Layout` folder. Think of adapters in Android as a mechanism to provide some customized formatting of elements or widgets within the user interface. All user interface components that use adapters, such as a `ListView`, use default adapters if you don't specify one explicitly, but you can always replace them with your own.

The final piece of the puzzle for the `FeedActivity` class is to override the `OnListItemClick` method so that we can open up a new `Activity` that shows us the actual content of the individual feed items that we touch.

```
base.OnListItemClick(l, v, position, id);
```

Once again, we call the base class method to be sure that all normal processing is being carried out.

```
var second = new Intent(this, typeof(WebActivity));
second.PutExtra("link", _items[position].Link);
StartActivity(second);
```

We now follow the Android design pattern for passing data to a new `Activity`. This will become very familiar to you as you create more applications that involve multiple screens. We create a new `Intent` object, which is the Android way of passing data to a new `Activity`. We pass it two objects that represent the context of where the call is originating, `this`, and the `Type` object to where it is going.

Once we have the new `Intent` object, we *put* things, typically strings, into it and pass it on. In this case, we use the `PutExtra` method to add a `key/value` pair to the `Intent`, and start the transition process to the `WebActivity` screen with the `StartActivity` method.

Based on the code involved in the creating of the `FeedActivity` screen, we now need to create a `FeedAdapter` class that populates and formats the `RssItem` data into our `ListView` and a `WebActivity` class to represent the next screen. Let's start with the `FeedAdapter` class.

```
public class FeedAdapter : BaseAdapter<RssItem>
{
    private RssItem[] _items;
    private Activity _context;

    public FeedAdapter( Activity context, RssItem[] items)
    {
        _context = context;
        _items = items;
    }

    public override RssItem this[int position]
    {
        get { return _items[position]; }
    }

    public override int Count
    {
        get { return _items.Count(); }
    }

    public override long GetItemId(int position)
    {
        return position;
    }

    public override View GetView(int position, View convert
    {
        var view = convertView;
        if (view == null)
        {
            view = _context.LayoutInflater.Inflate(Android
        }

        view.FindViewById<TextView>(Android.Resource.Id.Te
        view.FindViewById<TextView>(Android.Resource.Id.Te

        return view;
    }
}
```

Yikes. That's a lot of code. It's actually fairly simple though. We need to override four methods/properties on the base class, `BaseAdapter`. In our case, the generic parameter is going to be our `RssItem` class. The first three are fairly self explanatory.

- `this[int position]` returns an `RssItem` at the given position in the array.
- `Count` returns the number of `RssItem` objects in the array.
- `GetItemId` returns the `Id` of an `RssItem` at a given position, the position in our example.

The last, and slightly more complicated, override is the `GetView` method. This method gets an instance of the `ListView` within our `Activity` and `Inflate` it as a `SimpleListItem2`, which is a type of `ListItem` view within Android that allows two rows of text in a single item. We then set the first row of text to the `RssItem.Title` property and the second row of text to a concatenation of the `RssItem.Creator` property and the `RssItem.PubDate` property.

With the adapter set, we can focus on the second screen of our application, `WebActivity`.

```
[Activity(Label = "WebActivity")]
public class WebActivity : Activity
{
    protected override void OnCreate(Bundle bundle)
    {
        base.OnCreate(bundle);

        SetContentView(Resource.Layout.WebActivity);

        WebView view = FindViewById<WebView>(Resource.Id.De

        view.LoadUrl(Intent.GetStringExtra("link"));
    }
}
```

The structure is similar to the one of the `FeedActivity` class. We are once again using the `ActivityAttribute` to decorate the `WebActivity` class. There are only three slightly different lines in this method that we didn't encounter before.

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

The `SetContentView` method is a nice helper method that will map our C# `Activity` class to the specified layout file. In this example, we are referencing the `WebActivity.axml` file.

```
WebView view = FindViewById<WebView>(Resource.Id.DetailView
view.LoadUrl(Intent.GetStringExtra("link"));
```

The last two lines are specific to the `WebView` control within our layout. We use the `FindViewById` method to get a reference to the specified `WebView` control and call the `LoadUrl` method and pass it the data that was sent to this `Activity`, via an `Intent`, from the `FeedActivity` class.

The last pieces of the puzzle are the layout files that define the placing and naming of controls on the individual screens. The first one is the `Main.axml` file in the `Resources/Layout` folder in your solution. Simply replace its contents with the following:

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<LinearLayout xmlns:android="http://schemas.android.com/apk
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <ListView
        android:id="@+id/TutsFeedListView"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" />
</LinearLayout>
```

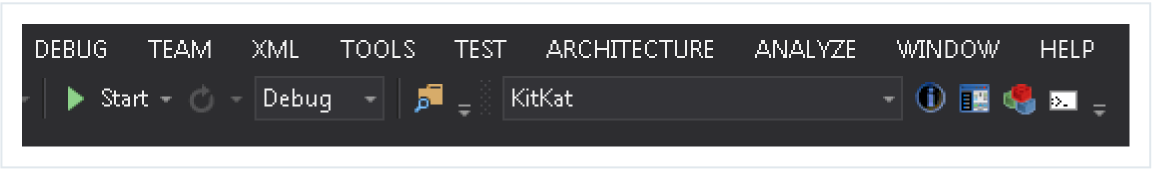
Next, create a new Android layout file in the `Resources/Layout` folder, name it `WebActivity.axml`, and replace its contents with the following:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
  <WebView
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:id="@+id/DetailView" />
</LinearLayout>
```

Build and Deploy

Once you've completed creating all the pieces of this application, you should be able to successfully build the application and deploy it to the Android Emulator. The build process is just like any other application you've created in Visual Studio. You can debug your application by pressing F5 or run your application using Control-F5. The only difference is that you have a number of deployment options you can configure. For this tutorial, we are interested in running the application in the Android Emulator, but if you have a physical device you can run your application on there as well.

In the Xamarin.Android toolbar, you have several different options. You have a drop-down menu that lets you specify the Android version the emulator should run. For this application, I have chosen to run on the latest version of Android, Android_API_19 or KitKat at the time of writing.



If you don't have the latest version of the SDK like I do here, you can open the Android SDK Manager and download the version you'd like to run your application on. If you open the Android SDK Manager, you can choose from a plethora of different Android SDK versions and a few additional tools.

You also have the option to configure the available emulators or create your own. This is done through the Android Emulator Manager in which you can create, edit, copy, and delete emulators.

By clicking the **New** or **Edit** button on the right, you are presented with a dialog through which you can configure the emulator.

Once everything's configured the way you like, it's time for the moment of truth, running your Android application. Press F5 and wait for the emulator to launch, which can take some time. It's therefore a good idea to leave the emulator open, so you don't have to wait for it to start up every time you deploy your application. Once your application is running in the emulator, you should see something like this.

Your view may differ slightly, depending on how you have configured the emulator. Tapping or clicking one of the titles should take you to a web view within your application that looks similar to the one below.

Conclusion

There you have it. You have successfully created an Android application using Visual Studio, C#, and a little help from your friends at Xamarin.

From here, you can take a number of steps. You can tailor this application to make it completely your own or leave it as is and impress your friends. Either way, you've taken a big step into the world of non-Microsoft mobile development using Microsoft tools. That's pretty cool in itself if you ask me.

Next time, we'll tackle the world of iOS development using Xamarin.iOS, which is a very similar process with only a few differences. Excited? I know I am. Until next time and happy coding.