# Converting a Xamarin.Android project to net7.0-android

**Selecting an App to Convert**

I chose my NavigationGraph2 project, which is part of a series of tutorials (https://github.com/gmck) about using Android’s NavigationComponent, Android’s recommended way of building modern Android apps –> Single Activity/Multiple fragments.

The reason I choose that project was that it was based on the NavigationDrawer App template from Xamarin.Android. This app template is probably the most used of the supplied app templates, and therefore a familiar app template to most Xamarin.Android developers.

**net7.0-android**

I’d never attempted to use net6.0-android, so I was surprised when Visual Studio defaulted to net7.0, rather than net6.0-android. The only detailed docs I could find were at <https://github.com/xamarin/xamarin-android/wiki/Migrating-Xamarin.Android-Applications-to-.NET-6>. I’m assuming that everything in the .NET-6 documentation applies to .NET-7. It suggests various ways of approaching the conversion, a try-convert and an upgrade-assitant, from a new template and lastly overwrite .csproj. There were warnings about the first two being incomplete and not fully supported, so I thought I’d check the new template which would then hopefully show what the new project structure was.

**To Begin**

I haven’t checked, but I believe that you will need to be using Visual Studio 17.4.1. I’m using the Preview version of Visual Studio 17.5.0 Prev 1.0.

So as usual select Create a new project and then choose *Android Application* – *a project for creating a .Net Android Application* (the first one after all the Maui templates). In the project wizard enter a project name and then on the next page enter the android package name and SupportedOSPlatformVersion which defaults to 21. The first surprise was the button at the bottom of this page is Create, therefore there are no other following choices like in a normal Xamarin.Android template. This single template is extremely basic. No theme, a MainActivity inheriting from Activity and the Standard mipmaps consisting of Xamarin’s new .Net app icons. That’s it – and no other templates, so we need to get more efficient in creating new projects. Either that or write our own templates.

Build and deploy the app just to make sure it works. Then study the new Application Properties and open the cs.prog file to view it as well.

Like all new apps I change the default namespace to com.yourcompanyname.navigationgraph2net7 etc. The Resource.Designer.cs no longer exists within the project, so you don’t have to modify that. It hasn’t disappeared completely – you will find it in the obj\Debug\net7.0-android\designtime folder. When you open it you will find that the global::Android.Runtime.ResourceDesignerAttribute and the namespace have been automatically changed, so that is a nice change.

Another file that has been removed is AssemblyInfo.cs.

I added <RuntimeIdentifiers>android-arm;android-arm64;</RuntimeIdentifiers> just for the devices I want to support because the properties window appears to only allow one entry. If you want to support other devices check the documentation for the new names. I’d suggest keeping the Application properties window and the Edit project window open while you are importing to observe the changes to the cs.prog file.

With the only change being the above, when I built and deployed with these changes, the MainActivity didn’t display full screen. I didn’t worry about it, because I am going to replace all of it anyway.

Both of the following are on by default. I’m experimenting setting them off (disable) because of the heap of problems they cause as you bring in your own code.

<Nullable>enable</Nullable>

<ImplicitUsings>enable</ImplicitUsings>

I’m not suggesting that you leave Nullable disabled, but I found it easier to get the application built without a ton of warnings. Warnings are always off-putting, and you’re inclined to want to fix them on the spot. Leave them until you are done with the conversion and then enable Nullable and go back and fix them. If you haven’t used Nullable before, then you’ll probably want to bookmark this link <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/compiler-messages/nullable-warnings>.

ImplicitUsings were a real pain. I just could not get them to work without creating extra work in modifying the code. In particular, by default you get Android.App. I’m not saying you will never use Android.App, you will, but when dealing with Fragments in modern Android apps you don’t want Android.App.Fragment, you always want AndroidX.Fragment.App.Fragment. Therefore you get a warning ***Fragment is an ambiguous reference between Android.App.Fragment and AndroidX.Fragment.App.Fragment****.* TheAndroid.App.Fragment was deprecated years ago.

The same happens in a Xamarin.Android classic app if you use Add Item and then choose Fragment. It defaults to Android.App.Fragment so you have green squiggles everywhere until you right-click Fragment and then it will suggest adding using Fragment = AndroidX.Fragment.App.Fragment; which gets rid of all the squiggles. However, that can be simplified further by removing the using Android.App which shouldn’t have been automatically added in the first place and then removing the “Fragment =” and just replace with a simple using AndroidX.Fragment.App;

Another point, there is no Add Fragment anyway. In fact, for Android, Add Item consists of 2 choices Android Activity template and Android Layout templates. You are going to have to install Android Studio if you want other layout templates. However, if you are doing Xamarin.Forms then you have access to 8 different templates.

I’m struggling to even see why I would use ImplicitUsings – someone suggested that you gain vertical space and therefore don’t have to scroll so far to find the code you want to modify. In NavigationGraph2Net7, MainActivity has 14 usings and each fragment has 4. Yeh, that is a lot of extra scrolling!!!. The other advantage of having the usings is that you can see at a glance what you are using and where a class comes from. When you are unsure of something you can comment out a single using and then look for the squiggles for why it was required. If you then see a using that is greyed out, then you know you can get rid of it, (right-click and *Remove and Sort Usings).* Having them there is an advantage in my opinion when you are learning something new.

So with those two settings disabled, let's get on with it.

First off, ditch all .Net mipmaps for your own. Just delete the contents and use add item to bring in your own. In this example, I‘m going to bring in mipmaps from my NavigationGraph2 project, which are the old black application Xamarin icons.

As you are doing this, keep the project window open and you will see all these resources appearing in the window.

<ItemGroup>

<None Remove="Resources\mipmap-anydpi-v26\ic\_launcher.xml" />

<None Remove="Resources\mipmap-anydpi-v26\ic\_launcher\_round.xml" />

<None Remove="Resources\mipmap-hdpi\ic\_launcher.png" />

<None Remove="Resources\mipmap-hdpi\ic\_launcher\_foreground.png" />

<None Remove="Resources\mipmap-hdpi\ic\_launcher\_round.png" />

</ItemGroup>

Next, do the same for the values folder. Delete all the contents and import all the contents from the NavigationGraph2 project. You’ll first need to create the following Resources folders *anim, drawable, drawable-v21, menu, navigation and xml*.

Now delete the contents of each folder i.e. the single layout in Layout and then use *Add Existing Item* and add the contents to each folder from the NavigationGraph2. project to NavigationGraph2Net7 project. The Resources folder in the NavigationGraph2Net7 project should now match the Resources folder of the NavigationGraph2 project when you compare them side by side.

We then need to create a Fragments folder off the root of the project. Once created then use Add Existing item and select all of the fragments in the NavigationGraph2 Fragments folder.

The last 3 files to add are AnimationResource.cs, NavFragmentOnBackPressedCallback.cs and MainActivity.cs. So delete the existing MainActivity.cs and then use *Add Existing Item* again to add those 3 files.

Add this stage your solution folder should look identical to the NavigationGraph2 solution. We still need to do some manual editing as the namespaces will not match. The new project is com.companyname.navigationgraph2net7, whereas the old project is just com.companyname.navigationGraph2. Notice that I now have adopted using all lowercase for namespaces since I wrote NavigationGraph2 (November 2021). You can either copy/paste the info to each file one at a time or use Find and Replace/ Replace in Files. Although I would suggest just using Find and Replace/ Find in Files first just to make sure that when you do use Replace in Files you are going to get it right.

One particularly sensitive file is the navigation/nav\_graph.xml. This is a case-sensitive file. The nav\_graph contains two important properties for each fragment. The android.id of each fragment is an int and corresponds to a menuItem.Id (how the NavigationComponent navigates) and the android:name of the fragment. The android:name corresponds to the fully qualified name of the fragment in the folder Fragments

For example the:

ImportFragment in NavigationGraph2

<fragment

android:id="@+id/import\_fragment"

android:name="com.companyname.navigationgraph2.fragments.ImportFragment"

android:label="Import"

tools:layout="@layout/fragment\_import" />

ImportFragment in NavigationGraph2Net7

<fragment

android:id="@+id/import\_fragment"

android:name="com.companyname.navigationgraph2net7.fragments.ImportFragment"

android:label="Import - Net7"

tools:layout="@layout/fragment\_import" />

Notice how the name of the class is capitalised because C# is case sensitive and our Class names begin with a capital. If it was entered lowercase here we would get an error class not found at runtime.

The reason why I’ve made the android:label different is just so you can recognise which apk you are running when you launch the apps as in the launcher they will look the same.

We are nearly done.

If you tried to build the project now you would get a heap of errors. We need to add the following NuGets using

Xamarin.AndroidX.AppCompat Version=1.5.1.1

Xamarin.AndroidX.Navigation.Fragment Version 2.3.5.3

Xamarin.AndroidX.Navigation.Ui Version 2.3.5.3

Xamarin.AndroidX.Preference Version 1.20.

Please note these aren’t the latest versions. It is important to keep them at these versions. If you read through the NavigationGraph.docx file which contains all the NavigationGraph projects, you will see where we do upgrade to the latest versions as the project number goes up.

In the solution under the **Dependencies** folder (a new folder in .Net7) you will find Packages. Right-click on packages and select Manage NuGet Packages – find each of these packages and select the above version numbers for each package and install.

You now should be able to build the project and deploy and run it successfully.

If you are deploying to a device running Android12 or above you’ll notice that when launching the app you get the automatic splash screen of the application icon. If the device is less than Android 12 you just get a white screen. We can fix that before we go any further, by installing the Xamarin.AndroidX.Core.SplashScreen Nuget and adding the line.

AndroidX.Core.SplashScreen.SplashScreen.InstallSplashScreen(this); before

base.OnCreate(savedInstanceState) in the MainActivity.

We also need to add the following in styles.xml.

<style name="AppTheme.Starting" parent="Theme.SplashScreen.IconBackground">

<item name="windowSplashScreenAnimatedIcon">@mipmap/ic\_launcher\_foreground</item>

<item name="windowSplashScreenIconBackgroundColor">@color/ic\_launcher\_background</item>

<item name="windowSplashScreenBackground">?android:colorBackground</item>

<item name="postSplashScreenTheme">@style/AppTheme</item>

</style>

The last change is to change the theme in the manifest to AppTheme.Starting ensuring that there is no theme reference in the Activity attribute of the MainActivity.

If you check the NavigationGraph2 project you’ll see that it was built with android:targetSdkVersion="31", before Xamarin.AndroidX.Core.SplashScreen was available hence it didn’t have a splash screen when deployed to devices less than Android 12.

Also, there was another gotcha with styles.xml. The original styles had the following theme which was set in the Activity attribute.

|  |
| --- |
| <style name="AppTheme.NoActionBar">. |
| <item name="windowActionBar">false</item> |
| <item name="windowNoTitle">true</item> |
| </style> |

We need to move those two item lines into the new AppTheme and then completely comment out AppTheme.NoActionBar.

The final change is to remove AppTheme.NoActionBar from the Activity attribute in the MainActivity, otherwise the app would crash immediately on SetSupportActionBar(toolbar) with

*Java.Lang.IllegalStateException*

*Message=This Activity already has an action bar supplied by the window decor. Do not request Window.FEATURE\_SUPPORT\_ACTION\_BAR and set windowActionBar to false in your theme to use a Toolbar instead.*

*I wish I had $100 for every time I’ve made that mistake since I started with Xamarin.*

The item postSplashScreenTheme in the styles.xml will automatically switch the theme to AppTheme when the splash screen finishes displaying,

One surprise I didn’t notice for a while, simply because I forgot to rotate to Landscape, was that the new app failed when rotated to Landscape after the Splash Screen was added.

Text

Description automatically generated

A typical screenshot in Landscape mode when using a device with a camera notch. What isn’t obvious from the screenshot is that you don’t see the black hole of the camera, but more importantly is that the toolbar or the AppBarLayout doesn’t extend to the full screen.

There is an easy fix for it.

I’ve seen this happen at various times during development before the Splash Screen API was available, but I wasn’t aware that by adding the Splash Screen API that it could happen.

To fix it insert the following code in OnDestinationChanged

// Include this if using the Splash Screen API

if (Build.VERSION.SdkInt >= BuildVersionCodes.P)

{

// Don't understand why we get this warning - when we are already checking //for >= BuildVersionCodes.P. //uncomment the pragma lines to see the

//SupportedOSPlatformVersion warning.

#pragma warning disable CA1416 // Validate platform compatibility

Window!.Attributes!.LayoutInDisplayCutoutMode = LayoutInDisplayCutoutMode.Default;

#pragma warning restore CA1416 // Validate platform compatibility

}

If you check my demo app XamarinBasicSplashScreen on my GitHub describing how to add the SplashScreen API to any Xamarin.Android app, you will see that it does the same thing, I just checked!!. I will have to fix that over the next couple of days. Since that app doesn’t use NavigationComponent the fix there is to add the same code in the OnCreate of the MainActivity.

The fix above also demonstrates a problem that I couldn’t solve when using <Nullable>enable</Nullable> in the project. If you haven’t used Nullable much in the past, then you are going to come across some tricky conversions. If you look at the code just above this where I struggled to come up with the correct implementation to cancel the warning.

navigationView!.Menu.FindItem(Resource.Id.import\_fragment)!.SetChecked(navDestination.Id == Resource.Id.import\_fragment);

I left the code I used (commented) to break it down into steps to come up with the final code and to show a technique of how you can sort it out. Until you get an understanding of Nullable, it can be difficult to get it right.

The other point I would like to raise is the use of #pragma warning disable CA1416 // Validate platform compatibility above the same code. I don’t understand why the compiler gives this warning when I’ve already wrapped the code in the correct condition Build.VERSION.SdkInt >= BuildVersionCodes.P. If you uncomment the pragma you get

**This call site is reachable on Android ‘24’ and later. WindowManagerLayoutParams.LayoutInDisplayCutOutMode is only supported on ‘Android’ 28.0 and over.**

If anyone has any ideas or solutions to that one, I like to hear them.

I discovered that you can use an attribute on the function to do the same thing.

[SupportedOSPlatformAttribute("android28.0")] which allows you to remove the pragma and CA1416 warning. However, that still doesn’t explain why either is necessary considering the condition.

**Release build**

All the above was done in debug mode. I wanted to check release apk sizes against the original NavigationGraph2 project as a comparison.

I forgot that I had a proguard.cfg file in the older app so it immediately crashed in release mode. I, therefore, had to add the proguard.cfg from the older app and that fixed the crash.

The older proguard.cfg included many more keep directives from my main app, so I modified this proguard.cfg to only include those keep directives that are applicable just for this app.

I also left comments that are useful for anyone introducing proguard.cfg for the first time into an app.

Note: When upgrading to later versions of the NuGets, some of these keep directives are no longer necessary because they are already included in NuGet, so you may not need the cfg file, so it’s a good idea to exclude the proguard.cfg when testing with later versions of the Nugets. If it then doesn’t crash you don’t need the proguard.cfg.

**Comparison of Release Signed aab**

NavigationGraph2 11,516KB

NavigationGraph2Net7 9,044KB

**Comparison of Release Signed apk**

NavigationGraph2 N/A

NavigationGraph2Net7 9,649KB

By default release builds use.

<PropertyGroup Condition="'$(Configuration)' == 'Release'">

<RunAOTCompilation>true</RunAOTCompilation>

<AndroidEnableProfiledAot>true</AndroidEnableProfiledAot>

</PropertyGroup>