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

The following are my observations regarding migrating several Xamarin.Android apps to .Net7 apps. Hopefully, if you follow these recommendations or tips, it should enable you to convert your app successfully to Net7 in a minimal amount of time.

I decided on the manual method (***from new template*** ) rather than the automated method ***try-convert*** or ***upgrade-assistant*** as mentioned in the following docs <https://github.com/xamarin/xamarin-android/wiki/Migrating-Xamarin.Android-Applications-to-.NET-6>.

It is a combination of the ***from new template*** and the ***Overwrite .csproj*** methods as by keeping the .csproj file open while you are doing the conversion you automatically see how the .csproj updates with each step and so learn more about the operation of how the new .csprog file works for future projects. There were also warnings about the first two methods being incomplete and not fully supported, so I thought I’d check the new template method and maybe try the other two methods later.

I ran into several problematic errors so hopefully, these notes will also be useful in your own conversions.

Rather than convert an existing published application, I would definitely recommend practising on any of the NavigationGraph projects that I’ve made available at <https://github.com/gmck> or on any of your test projects, before attempting to migrate a published application.

In my own case, I’ve only converted test apps that were already converted to AndroidX and were already targeting Android 12 or Android 13. If you haven’t reached that point, converting to AndroidX then I would recommend that you complete that step using Xamarin.Android before migrating to .Net7.

I was fully expecting to start with .Net6, but as the default ***new template*** starts at .Net7. I decided to ignore .Net6, assuming that potential problems with .Net6 would have already been corrected in .Net7.

**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 (just accept the default), we aren’t going to do anything with this project) 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 will need to get more efficient/creative 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 (view Edit Project File) to view it as well. The project file is the most interesting and is tiny compared to a normal Xamarin.Android project. The application Properties window definitely needs to be improved and no doubt it probably will be in future versions. This is the responsibility of the Visual Studio team, not the Xamarin.Android team. The application Properties window is also a little flaky so I wouldn’t spend a lot of time there. For instance, I wanted to change the Platform target, but it will only allow a single choice, which for an android app would default to all – not what I wanted. I quickly lost interest there, so I went to the docs to find out how.

I’m also interested in getting a couple of properties set correctly even before I write a line of code. So, in any project I create I’m going to want to set the following, in addition to what is already there in the first property group.

ApplicationId>com.companyname.navigationgraph6net7</ApplicationId>

<RootNamespace>com.companyname.navigationgraph6net7</RootNamespace>

<RuntimeIdentifiers Condition=" '$(Configuration)' == 'Release' ">android-arm;android-arm64</RuntimeIdentifiers>

And then create a new property group for release builds only. This will ensure that for debug builds we are always using fast deployment.

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

<AndroidLinkTool>r8</AndroidLinkTool>

</PropertyGroup>

Generally, Xamarin has always done a reasonable job when using pascal casing convention (C# default) for namespaces, concerning Android’s use of lowercase but in many situations, such as using the NavigationComponent (single activity/ multiple fragments), so to me this is just a simple step that guarantees there are no collisions requiring workarounds when your app eventually reaches Google Play.

These changes are simpler to make in a .Net project as compared to Xamarin.Android. You’ll probably also notice that the new project doesn’t have a Resource.Designer.cs file or an AssemblyInfo.cs file. The Resource.Designer.cs has been moved to obj\Debug\net7.0-android\designtime\ Resource.designer.cs and if you open it, you will find that Runtime.ResourceDesignerAttribute and the namespace have both been changed correctly, which is an improvement on a Xamarin.Android app where they had to be manually changed.

Another file that has been moved is the AndroidManifest.xml file. The Properties folder has been dropped so the android manifest file is now in the root folder of the project. The AssemblyInfo.cs is now in obj\Debug\net7.0-android with different content.

**Both Nullable and ImplicitUsings are enabled by default**

<Nullable>enable</Nullable>

<ImplicitUsings>enable</ImplicitUsings>

Both default settings **ImplicitUsings** and **Nullable** create problems when migrating an existing app.

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*** when creating a new Fragment*.* TheAndroid.App.Fragment was deprecated years ago.

The same happens in an Xamarin.Android classic app if you use Add Item and then choose Fragment. It defaults to Android.App.Fragment so you have green squigglies 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 .Net7 project. In fact, for Android, Add Item consists of 2 choices Android Activity template and Android Layout templates, both layout templates – no other templates of any type. 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 can’t help but think that Microsoft thinks every C# Android programmer wants to use Maui, when quite clearly there are many of us who don’t have that requirement and don’t need or want Maui, if we only want to develop apps for Android devices.

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 especially when you are learning something new.

**<Nullable>enable</Nullable>**

I’m not suggesting that you leave Nullable disabled, but I found it easier to get the application built and running without a ton of warnings. Warnings are always off-putting, and you’re inclined to want to fix them on the spot. My suggestion is to 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>.

So, with those two settings disabled, let's get on with it. You can either use one of your own projects or download one from <https://github.com/gmck>. I’d suggest using <https://github.com/gmck/NavigationGraph6> and then you can proceed by following the steps below.

Download NavigationGraph6 which is a standard Xamarin.Android Classic project and unzip to any test folder. Build it and deploy it just so you can see what it looks like. It doesn’t really matter if you don’t have any experience using the Google’s NavigationComponent as that is not the point of this particular exercise. We just need a project to be able to convert so the steps of the conversion can be understood.

Assuming you have now done that, then we need to create a new .Net7 project. To make it comparable you should create a new project using *Android Application* – *a project for creating a .Net Android Application* just as you did before with AndroidApp1, but this time name the project **NavigationGraph6Net7,** so that we can distinguish between the two apps on the device. This way both projects will end up on your device, and unfortunately, because of the long names, I chose originally they will be difficult to distinguish between unless you allow rotation of your home screen.

Your project file should look like the following once set up.

<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>

<TargetFramework>net7.0-android</TargetFramework>

<SupportedOSPlatformVersion>24</SupportedOSPlatformVersion>

<OutputType>Exe</OutputType>

<Nullable>disable</Nullable>

<ImplicitUsings>disable</ImplicitUsings>

<ApplicationId>com.companyname.navigationgraph6net7</ApplicationId>

<ApplicationVersion>1</ApplicationVersion>

<ApplicationDisplayVersion>1.0</ApplicationDisplayVersion>

<RootNamespace>com.companyname.navigationgraph6net7</RootNamespace>

<RuntimeIdentifiers Condition=" '$(Configuration)' == 'Release' ">android-arm;android-arm64</RuntimeIdentifiers>

</PropertyGroup>

</Project>

I would suggest making these settings directly via the .csprog file, rather than using the Properties page as mentioned above. You could just simply copy/paste the above and overwrite the contents of the newly created .csproj file. ***Make sure you can build and deploy the app before proceeding further.***

The next step is to include all the AndroidX Nuget packages that the NavigationGraph6 project was using. It is important to use the same version numbers as NavigationGraph6 was using, so you will have to select each one individually when installing them.

Alternatively, rather than install them individually by selecting the correct version etc you could just copy/paste the ItemGroup below to the end of the project inserted before the final </Project> tag.

When completed your csproj file should have the following item group added to the bottom of the file

<ItemGroup>

<PackageReference Include="Xamarin.AndroidX.AppCompat" Version="1.5.0" />

<PackageReference Include="Xamarin.AndroidX.Core.SplashScreen" Version=" 1.0.0.1-rc01" />

<PackageReference Include="Xamarin.AndroidX.Navigation.Fragment" Version="2.5.1" />

<PackageReference Include="Xamarin.AndroidX.Navigation.UI" Version="2.5.1" />

<PackageReference Include="Xamarin.AndroidX.Preference" Version="1.2.0.2" />

<PackageReference Include="Xamarin.Google.Android.Material" Version="1.6.1.1" />

</ItemGroup>

A third way of achieving the same result (say for your own projects), would be to open the original NavigationGraph6 project and then unload the project and chose Edit the project file. Find the Packages in the project and copy/paste them directly into the new project. This would obviously be the quickest way, however, the syntax of the old project requires a slight change to match the new one-line entry of the .Net7 format, so some minor editing of each PackageReference is required.

As a first step, we want to add all our Resources. So before doing any adding we need to create all the folders to duplicate the folder names of the NavigationGraph6 project. So, you need to select the Resources folder and Right click and select Add/New Folder and repeat adding the missing folder names until the folder structure of NavigationGraph6Net7 matches the folder structure of NavigationGraph6. Notice how as you add the new folders the new project file updates with the new entries.

We are now ready to start importing the contents of each Resource/folder of the NavigationGraph6 project. Start with the ***anim*** folder (the equivalent folder) in the NavigationGraph6Net7 project right click on ***anim*** and select Add/Existing Item and navigate to the ***anim*** folder in NavigationGraph6 and select all the items. Note how the project folder of NavigationGraph6Net7 is modified as the items are imported.

Now theoretically we do the same with each folder until we have completed all the folders with matching contents in both projects. Obviously, with the layout folder, we first delete the original activity\_main.xml, because we are replacing not adding to the layout folder. Normally we would do the same with the mipmap folders, but I would suggest that in this exercise your leave the .Net7 mipmaps, just to make it easier to distinguish between the two apps on the device. The values folder also needs strings.xml to be deleted before adding the NavigationGraph6 projects strings.xml.

Unfortunately, when we come to import the contents of the drawables folder we hit a bug with the new project format. As a result of this bug, before adding the drawables we need to make a further change to the project before importing any of the \*.webp files in the drawables folder.

Support for webp files was forgotten by the Xamarin.Android team, so the build mechanism was missing the ability to handle support for importing webp files. I came across this bug when converting NavigationGraph7 to NavigationGraph7Net. I reported the bug to the Xamarin.Android team and they quickly came up with several workarounds, that fixed the problem - you can follow the short-term and long-term solution here <https://github.com/xamarin/xamarin-android/issues/7600>. To fix it while waiting for their solution I just converted the webp files to jpg which the other project accepted.

So, before we proceed to add the contents of the drawable folder, we need to insert the following in the project file.

<ItemGroup>

<AndroidResource Include="Resources\\*\\*.webp" />

</ItemGroup>

This should be inserted after the end tag of the first PropertyGroup. Once inserted and the file saved, you can then successfully add all the contents of the drawable folder.

You also need to consider the contents of the Resources folders. For instance, do they have anything in them that is specific to the particular project that they were created for? For instance, a couple of the NavigationGraph6 resources have references to namespaces. Having the wrong namespace at this point would not prevent the app from building successfully, because we don’t have any code yet that references a namespace. However, as we add code, we will certainly come across build errors because of those now incorrect namespaces. Whether you fix them now, or when they occur during a build is up to you, but you could use the search facility of Visual Studio *Find and Replace/ Find in Files* to search for “navigationgraph6” and note where changes will be required. Just so you know ahead of time, changes will be needed in preferences.xml and nav\_graph.xml.

Assuming, that the project now builds without error, you can then proceed to importing all the code.

Again, we need to mimic the folder structure of the NavigationGraph6 project. So, we need to create 3 new folders Adapters, Fragments and Dialogs before we can begin to import the code. Once they are created. The process of migrating the project should be as simple as just selecting the folder and using Add/Existing item to each folder and importing every file for each folder from the NavigationGraph6 project. The same then applies to the .cs files in the root folder of the project*. Just remember to first delete the original MainActivity.cs.*

While doing all of the above, make sure you also have the new cs.proj file open to see how the cs.proj works so you gain an understanding of how to build or edit a cs.proj file manually in the future.

Finally, the androidmanifest.xml. Copy/Paste from the old to the new and add ‘net7’ (without the quotes) to the end of the packagename.

For now, we are done with the migration part, it is probably a good time to look at the project structure of the two cs.prog files and appreciate the improvements that have been made in the project structure of a Net7 project. The old project was 291 lines, the new project is 79 which would have been reduced by a couple more lines if not for the problem of the webp drawable files. This indicates that the Xamarin.Android team have come up with very desirable defaults to simplify the set-up of both debug and release builds, while still allowing the developer to add specific features to their builds. To read more see the following https://github.com/xamarin/xamarin-android/blob/main/Documentation/guides/OneDotNet.md

**TODO: Replace namespaces**

If you’ve got this far, you probably already know the next step. We now need to fix all the namespace problems that we would encounter if we built the project now. You probably should build the project just to see all the errors.

You can either open each file and manually change *NavigationGraph6* for *navigationgraph6net7* with a simple copy/paste. That is the super safe way but obviously the slowest. Or use Visual Studio’s Find and Replace/Replace in Files. If you do go for the Replace in Files option, I’d suggest that you first do a trial run Find and Replace/Find in Files to see what the replacements will be.

A note about nav\_graph.xml and namespaces. The NavigationComponent was the main reason for adopting all lowercase re namespace, applicationids etc. If you open the nav\_graph.xml file, you’ll note that each fragment has an android:id and an android:name.

<fragment

android:id="@+id/home\_fragment" android:name="com.companyname.navigationgraph6net7.fragments.HomeFragment"

android:label="@string/menu\_home"/>

The id is an int and corresponds to a menuItem.Id which is how the NavigationComponent navigates. Check the MainActivity and you will see the OnNavigationItemSelected(IMenuItem menuItem) is very similar to any other Xamarin.Android app. So the fragment is associated with a menuItem.Id via the navgraph.

The android:name is the actual name is the class name of the Fragment that is instantiated when that menuItem.Id is selected. Note that the name of the fragment is cased just like it appears in the code HomeFragment.cs. The important part is the constructor of the Fragment which is automatically called by Android during initiation or on rotation. That is the main reason that it must be a parameterless constructor because it is automatically called by the framework on a configuration change. To keep in sync with android in general it became far simpler to lowercase all the namespaces requirements and just keep this one rule for the Navigation Component of uppercasing the name of the fragment for C# requirements. I came to that conclusion after getting too many class not found exception crashes at start-up because of misspellings using the wrong case. The same applies to the two classes SystemThemeListPreference and ColorThemeListPreference found in preferences.xml.

One further change was made to accommodate the new Net7 icons. As is, when displayed by the splash screen the icon has a black background, which doesn’t look good. If you check the ic\_laucher\_background.xml in the Values folder you will find that it is still set to the colour of the old Xamarin icon, so I changed it to the colour extracted from the icon.

After some additional advice from the Xamarin.Android guys, I also introduced the following for release builds. This is inserted after the end tag of the first PropertyGroup.

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

<AndroidLinkTool>r8</AndroidLinkTool>

</PropertyGroup>

**TODO: Change to <Nullable>enable</Nullable>**

There isn’t much point in supplying a test app already converted using <Nullable>enable</Nullable> i.e. a build without warnings as that is not going to teach anyone how to do it, so this sample is posted as is with <Nullable>disable</Nullable>.

I’m struggling to think of a way to instruct someone how to use Nullable enabled via written instruction. The only practical way would be by example using a video, however, I don’t have the resources or the time to do that.

The best I can come up with is to consider an example. Take for example the class variables of the MainActivity.

private AppBarConfiguration appBarConfiguration;

private NavigationView navigationView;

private DrawerLayout drawerLayout;

private BottomNavigationView bottomNavigationView;

private NavController navController;

private MaterialToolbar toolbar;

They are declared here because obviously, they are going to be used in various methods of the MainActivity and since they haven’t been assigned any values and don’t have default values they are null and potentially could be attempted to be used while they still are null and would therefore fail.

However, if we declare them as null then the IDE is going to warn us each time we access the variable. So, we do that by changing the declarations with the ? null operator

private AppBarConfiguration? appBarConfiguration;

private NavigationView? navigationView;

private DrawerLayout? drawerLayout;

private BottomNavigationView? bottomNavigationView;

private NavController? navController;

private MaterialToolbar? toolbar;

However, that doesn’t consider code from the framework or any of the libraries we have used.

Again, from the MainActivity, let’s consider how the OnCreate override is written. OnCreate is from the Class Activity.

protected override void OnCreate(Bundle savedInstanceState)

{

}

However, in this instance, it is not because our MainActivity inherits from our BaseActivity.

So, we really need to know how the OnCreate() parameter Bundle is written. So, we need to go to its definition. Right-click Go to Definition or Peek Definition, then the same thing on base.OnCreate and then again from BaseActivity we need to keep going up the inheritance chain until we find it in the Activity class.

*protected virtual void OnCreate(Bundle? savedInstanceState);*

Here we find Bundle is declared with ?. i.e. can be null. Therefore, to get rid of all the warnings, we need to correct the BaseActivity and MainActivity by applying ? to Bundle.

As you can see this can be rather tedious when converting code. I haven’t written any code in a new project yet with Nullable enabled, but I’m presuming the IDE is going to warn me earlier when I write incorrect non-nullable code if the project is set using <Nullable>enable</Nullable>.

As tedious as it appears to be, I don’t see any other way of correcting existing code, unless one of the migrations tools mentioned earlier is capable of this.

Let’s take another example using the HomeFragment as an example. What we are required to do with the HomeFragment re nullable applies to all the other fragments. So fix the HomeFragment and then apply the same fixes to the others and you just eliminated say 40 odd warnings.

If you open the HomeFragment with nullable enabled you will find lots of green squigglies. Like every other fragment, the HomeFragment has on OnCreateView, so let’s look at OnCreateView by Go to Definition or Peek Definition on base.OnCreateView. You will see that both ViewGroup and Bundle have the ? operator. Therefore, to fix the nullability problem we need to add the ? to both ViewGroup and Bundle of the OnCreateView of the fragment. That gets rid of the squiggly on OnCreateView. The next line declares View view, therefore since inflator.Inflate refers to container which could be null, we have to declare View? view, as view could be returned by inflator.Inflate as null.

Similarly, the following line

TextView textView = view.FindViewById<TextView>(Resource.Id.text\_home);

Then becomes:

TextView? textView = view!.FindViewById<TextView>(Resource.Id.text\_home); // ! means maybe null

Note if you hover over the next line

return view – you get a message can’t be null, so view on that line doesn’t need the !.

With those changes – no more green squiggles in the OnCreateView.

We will make a couple more changes. The constructor of HomeFragment has green squigglies. If you hover over the constructor Visual Studio will show the warning message CS8618 (remember the link provided above re nullable) telling you that the two variables onBackPressedCallback and menuHost should be declared as nullable. So as soon as you add ? to each the squigglies go away on the constructor. So that is how you have to proceed through all your code when you enable Nullable.

Another alternative to Visual Studio’s Right click Go to Definition or Peek Definition is to use DotPeek to load the Nugets.

If you enable Nullable on this project e.g. <Nullable>enable</Nullable> you get 186 nullable type warnings. That gives you some idea of the amount of work that you will be required to do to get a clean build. This project is tiny as compared to my published projects and therefore I would expect anyone who has a published app would be thinking the same – how long is this going to take?

Another way of hiding all the warnings is to add **#nullable disable** to the top of any file. So you could take it file by file and then only have to contend with warnings generated by the remaining file that doesn’t have the #nullable disable at the beginning of the file.

Having done these conversions a couple of times now I can assure you it does get easier as you get more familiar with the experience and I’m afraid the only way to speed it up is to get that experience, so you recognise each error as you see it and can correct it quickly.

If you do get stuck on a particular piece of code, I’ve got a cheat sheet for you.

The NavigationGraph7Net7 project is already converted to nullable. The Navigation7Net7 project is based on the NavigationGraph6 project. The NavigationGraph7Net7 project is the same except that it is a Material 3 themed app as compared to NavigationGraph6 being a Material 2 themed app. It has the same structure as the NavigationGraph6 project with a couple of extra Fragments just to demonstrate the Material 3 components. Therefore, open the equivalent file in the NavigationGraph7Net7 app to see how a particular nullable problem was solved.

It even includes some of my more laborious efforts where I had to break a single line into multiple lines just to understand what I was attempting to do. I’ve left the commented multi-line code version in the code so you can see how I worked it back to the original single line of code.

You’ll also see where I got a touch fed up when converting a Toast message. Turning a one-line Toast into a three-line nullable version didn’t exactly inspire me, so I hit it with the big hammer (#pragma warning disable CS8602 / #pragma warning restore CS8602).

**One of the warnings requires some explanation**

The following code in the SettingsFragment produces an unexpected warning

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

uiModeManager?.SetApplicationNightMode((int)uiNightMode);

Warning CA1416 This call site is reachable on: 'Android' 24.0 and later. 'UiModeManager.SetApplicationNightMode(int)' is only supported on: 'android' 31.0 and later.

The code is perfectly acceptable Android code, which you would see in any Android application. It is required because uiModeManager.SetApplicationNightMode() is only available on Android 12 and above. The code has already been corrected to account for nullability, yet we still get a warning.

Why the warning?

There is also similar code in the same fragment which doesn’t raise a warning

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

{

systemThemeListPreference.Init();

systemThemeListPreference.PreferenceChange += SystemThemeListPreference\_PreferenceChange;

}

else

systemThemeListPreference.Enabled = false;

Why then, no warning for this line?

The first warning and its fix are explained here in this now-closed issue <https://github.com/xamarin/xamarin-android/issues/7586>.

I don’t have an explanation for the second example, I can only assume that because it doesn’t call a function that is potentially tied to a method of a class that requires the condition then it is not required.

Therefore instead of using the code in the first example, we are now required to code it as

if (OperatingSystem.IsAndroidVersionAtLeast(31))

uiModeManager?.SetApplicationNightMode((int)uiNightMode);

Right-clicking on OperatingSystem. IsAndroidVersionAtLeast(31) explains where it comes from.

If you have any questions you can contact me via email - gmckechnie1246@gmail.com.