May 19th 2022.

**Android’s NavigationComponent**

This solution contains several projects describing features of Android's new architecture known as the NavigationComponent. It is also known as a single activity/multiple fragment application. Each project introduces different techniques adding more complexity as the project number increases.

Some of the benefits of the Navigation Component include:

Automatic handling of fragment transactions

Correctly handling of the up and back buttons by default.

Default behaviours for animations and transitions

Implementing navigation UI patterns as in navigation drawers and bottom navigation with little additional work.

The major components are a Navigation Graph an XML resource, a NavigationHostFragment, a container, which holds each of the Fragments that make up the application and the NavController which maintains information about all the fragments and is responsible for navigating from one fragment to another.

**NavigationGraph1 Project**

This project is based on the existing standard Xamarin Navigation Drawer App template. If you want to create your own project, and I do recommend that you do, you can start with this. More on why later.

However, you will find that we basically strip out most of the code from the MainActivity. The only code left with be SetContentView and a couple of lines in the OnCreate. The default Navigation Drawer app doesn’t have any fragments connected to any of the menu choices, in fact, it doesn’t do much at all. It only has two xml layouts, activity\_main and content\_main. We will retain most of activity\_main except for the FloatingActionButton which can be deleted. (Please note there is nothing wrong with having a FloatingActionButton, I just decided it wasn’t adding any functionality in this example and therefore decided to remove it.)

content\_main will be completely replaced by the content\_main in NavigationGraph1. This layout contains an androidx.fragment.app.FragmentContainerView inside a androidx.constraintlayout.widget.ConstraintLayout.

Since we’ve introduced the FragmentContainerView we, therefore, need to revise the Nuget packages that are used. You should remove Xamarin.Essentials (we don’t make use of it) and Xamarin.Google.Android.Material. Then update Xamarin.AndroidX.AppCompat to 1.4.1. You then need to add two new packages Xamarin.AndroidX.Navigation.Fragment and Xamarin.AndroidX.Navigation.UI. Both these versions should be 2.3.5.3. Xamarin.AndroidX.Navigation.UI has a dependency on Xamarin.Google.Android.Material version 1.4.0.4), therefore there is no need to add Xamarin.Google.Android.Material as a separate package.

The reason I have limited the version to 2.3.5.3 is to avoid a bug that is contained in Xamarin.Google.Android.Material in version >= 1.5.0. The bug involves the BottomNavigationView which is not in this project but is involved in NavigationGraph3, so I’ll expand on that bug when we get there.

If you are building the app from the standard Navigation Drawer App template, before you go any further I would recommend that you make the following changes. Select the project, select Properties and when the properties window opens select Application, the top tab and then change both Assembly Name and Default Namespace to NavigationGraph1 and com.companyname.NavigationGraph1 respectively. You will then need to open Resource.Designer.cs and change the first two lines inserting the new default namespace of the file to the following

[assembly: global::Android.Runtime.ResourceDesignerAttribute("com.companyname.NavigationGraph1.Resource", IsApplication=true)]

namespace com.companyname.NavigationGraph1

Save Resource.Designer.cs. Now open MainActivity.cs and change the namespace to match. Clean the solution and rebuild. Feel free to change “companyname” to your actual company name. This will also change the package name of the application and should be reflected in the AndroidManifest.xml

The reason for this change now is that we have to add a Navigation Graph to the project and then add some fragments in a folder called Fragments. The navigation component is very picky about resource.ids and class names. Get any class names wrong and you’ll end up with a crash - class name not found.

So, you need to create a new folder off the root of the project called Fragments. Each fragment class will be created inside that folder and the namespace of each class must be the new namespace com.companyname.NavigationGraph1.Fragments. This should be the default when you create a new Fragment via the Wizard - add new item for a fragment. To write this I’m using VS 2022 17.2.0 Preview 5.0 and this wizard appears to be broken because it is not inserting the modified namespace automatically, which it did do correctly in previous releases. The other problem with the wizard is that it still thinks it is 2019 and the latest version of Android is Android 9 and so it adds an Android.App.Fragment instead of an AndroidX.Fragment.App. So much for the wizard, it is quicker and easier to do it manually.

To make it easier, please just add the fragments and their xml layouts by using Add Existing item and locating the respective folders in this project.

Once you have done that, we still need to implement the navigation graph. First, create a new folder off the Resources folder called navigation. This will hold the file nav\_graph.xml. Then you need to create a new xml file called nav\_graph via right-clicking on the navigation folder/Add new item and choosing xml file.

Copy/paste the contents of the existing nav\_graph.xml from the supplied project NavigationGraph1 into your new nav\_graph.xml. The following are the important points of a navigation graph. All navigation graphs have to have a Starting Destination, or top-level fragment as in app:startDestination="@+id/import\_fragment".

They also must have an android:id="@+id/navigation". Which is used within the FragmentContainerView as in app:navGraph="@navigation/nav\_graph"

Each fragment of the application is then listed as

<fragment

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

android:name="com.companyname.navigationgraph1.fragments.ImportFragment"

android:label="Import"

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

The android:id is how the fragment is linked to the android\_main\_drawer.xml file for the NavigationView. Notice how the menu item android:id is the same as the fragment’s android.id.

This provides the link for automatic navigation. The android:name attribute is the fully qualified fragment class name, so internally it knows which class to instantiate. The android:label is the string that appears in the top app bar of each fragment. The tools:layout can be disregarded as the Xamarin.Android designer among other things doesn’t know how to display the layout of the fragment, so that attribute is only applicable to Android Studio and can be omitted.

Notice how the android\_main\_drawer.xml has extra menu:ids nav\_share and nav\_send. We don’t have fragments for these two, nor layouts etc. However, they don’t cause a crash, because there is nothing to link the menu:id to a fragment. Therefore all that happens when you click on either of those menu items, is that the item will be highlighted. This is a very useful feature that allows you to build your menus etc to prototype an app without the need to complete all the code. It, therefore, acts as an effective TODO list.

I’ve used the naming convention of reversed layout name for the android:id, but feel free to adopt whatever naming convention you like.

One important Xamarin consideration is the android.name. It must match re case the actual fragment name as in ImportFragment. The rest of the package name, namespace com.companyname.navigationgraph1.fragments can be in lower case, but the actual fragment class name must be capitalised. Get this wrong and you can then expect to see *Unable to instantiate fragment* when it crashes. The best idea is to stick with lowercase namespace names. But the main point is that the Fragment name should be capitalised to match the code in your ImportFragment.cs file.

If it was not capitalised, the following would result - Unable to instantiate fragment com.companyname.navigationgraph1.fragments.importFragment.

You should now be ready to build a debug version and deploy it. If I’ve left any instructions out, just run NavigationGraph1 from the supplied project. Please let me know what I left out and I’ll revise this doc.

Conclusion:

If running successfully the app should respond by swapping fragments for each of the first 4 menu items. Notice how the ImportFragment (you might regard it as the home fragment ) ie. the StartDestination fragment is the only fragment that has a 3 dot menu for options, the other three fragments have deliberately had the menu cleared. Notice how the back button or back gesture always returns the user to the ImportFragment and any further back button or back gesture exits the app. This is an extremely easy method of navigation, whether using a back key or back key gesture and something all users will find very consistent when compared to any Google app such as Gmail.

There are a couple of points that are worthwhile discovering just by using the debugger that helps to explain the behaviour of the NavigationComponent, but first, note the addition to the MainActivity is that we have added the interface NavController.IOnDestinationChangedListener and its corresponding event handler OnDestinationChanged. As yet there is no code within the event handler, that will be introduced in NavigationGraph2.

Set a breakpoint on SetContentView. You will notice as you step that it will step to the constructor of ImportFragment, then the OnCreate of the ImportFragment and then if you keep stepping it returns to MainActivity’s OnCreate and then proceeds to the end of OnCreate. The next step will take you to OnDestinationChanged. Stepping out of OnDestinationChange leads back to MainActivity’s OnCreate and the next step takes you to ImportFragment’s OnCreateView to inflate its layout. The important point is that OnDestinationChanged is called even before the fragments OnCreateView method. This is quite a radical change from a conventional Multi-Activity app.

The other important method to note in MainActivity is OnSupportNavigateUp(). If you comment out this method, then tapping the hamburger does nothing which means, you can’t navigate because the nav drawer menu isn’t visible.

A more subtle change is contained within OnOptionsItemSelected. Note the return line. We will explore that change in NavigationGraph2.

One feature it doesn’t have is any animation of the fragments as they are changing. You may have read about adding animation via the nav\_graph, which of course can be done. However, I believe, that with a little extra code, we can control the animation of the fragments and give the option of different animations as a user choice, which I don’t believe can be obtained just by using a nav\_graph.

WARNING:

One thing you are going to strike is an incompatibility between the Xamarin.Android designer and the nav\_graph.xml For some reason the Xamarin.Android designer can’t handle a nav\_graph.xml. It may not happen immediately, but I can assure you it will eventually. When you open any xml layout file, the layout designer window will crash, but the xml window will still be viewable and editable.

I first reported this crash back in Feb 2020 and it is still not fixed – now over two years later, so you just have to learn to live with it. The designer window will open correctly if you exclude the nav\_graph from the project, which sort of proves the point that the designer can’t handle a nav\_graph. Of course, if you exclude the nav\_graph the app will not build, so you must remember to include it again after viewing the designer window before it will build correctly. I can assure you that quickly becomes a real pita.

There are a couple of workarounds, that makes giving up on the designer easier. Android Studio has a superb designer that easily does the job. The downside of using it is that you then have to maintain a project in Android Studio with all your xml layouts matching exactly the layouts in your Xamarin.Android project. That means copying and pasting from Android Studio to an empty layout of the same name in Visual Studio. You need to be very disciplined to maintain that over a long period. A second alternative is to use Beyond Compare and instead of copy/pasting open both xml files in BC and make new changes in Android Studio and then push them to Visual Studio. That is workable.

Another alternative I tried recently was to download JetBrains Rider. Load your Xamarin.Android solution/project into Rider and all the designer problems are solved. The disadvantage here is that you are virtually giving up on Visual Studio just so you can use Rider. They are compatible so the same solution/project can be opened in either, but using Rider also means learning a new IDE.

Let’s face it, the Xamarin.Android designer sucks. It hasn’t been updated in functionality since 2019 so I don’t expect it to change any time soon. It will never have the sophistication of Android Studio so does it matter? Consider trying to layout a ConstraintLayout in Xamarin.Android designer, you literally would know where to start.

After reporting the problem on Xamarin.Android Github in Feb 2020 I was eventually asked to report the problem using Visual Studio’s Send Feedback/Report a Problem as the Xamarin.Android team aren’t responsible for a Visual Studio bug. So late last year I eventually used that mechanism, explained what the problem was, supplied them with a test app to demonstrate the problem and in February received a reply from a bot stating the problem would most likely not be rectified, because my complaint/bug didn’t receive enough votes. Hard to believe that Microsoft is so pathetic when it comes to fixing bugs. Bugs get fixed by users voting…. That’s probably good to know because using their Send Feedback/Report a Problem system would appear to be a total waste of your valuable time.

**NavigationGraph2 Project**

We said above that to use animations between fragments you can add them to the nav\_graph. However, they are added via an action tag. Use an action tag inside the fragment tag and implies that you also provide the id of the fragment to which you want to navigate to as in

<action android:id="@+id/action\_fragmentA\_to\_fragmwentB"  
                app:destination="@id/b"  
                app:enterAnim="@anim/nav\_default\_enter\_anim"  
                app:exitAnim="@anim/nav\_default\_exit\_anim"  
                app:popEnterAnim="@anim/nav\_default\_pop\_enter\_anim"  
                app:popExitAnim="@anim/nav\_default\_pop\_exit\_anim"/>

This translates to navigate from destination fragment A to destination fragment B. Then before the closing tag the enterAnimation, exitAnimimation and the popEnterAnimation, popExitAnimation are assigned. When I came across this technique the first time I viewed the NavigationComponent I was put off because I just didn’t like the idea of having all this xml just to control how one fragment navigates to another fragment. As we saw in the NavigationGraph1 project, the following 2 lines of code

NavigationUI.SetupActionBarWithNavController(this, navController, appBarConfiguration);

NavigationUI.SetupWithNavController(navigationView, navController);

Which handled all the navigation automatically, based on whatever fragment:ids were put in the top-level fragment arraylist. We navigated to the fragment we wanted to go to by selecting the appropriate menu choice. So what was lacking was animating the change. The above shows one way of adding animations, but the disadvantage is that it appears you have to settle on the one type of animation for all the fragments.

I’ve got no objection to having to maintain a nav\_graph in my app as long as it just consists of a list of fragment tags. Very easy to maintain with a simple text editor. If you need a new fragment, add it in the nav\_graph as we have already done, add the Fragment.cs file and code your fragment and its layout and then hook up the fragment ids to the menu ids and it should all work. However, I dislike the idea of adding animation via an action. Actions in my opinion are not readable. I’d much rather read code to follow how an app navigates than have read a navgraph and its Actions.

The NavOptions Class, among other features, can set up Animations. Therefore if we can control animations through NavOptions I would prefer to do it that way even if I have to write a little more code. So we added a class AnimationsResource which allows you to define particular animation sets.

It has the advantage of being able to define whatever type of animation you require – and as many animations as you want. The two I’m demonstrating in NavigationGraph2 are Fader2 (my default) and Slider. Fader2 is very subtle whereas Slider is quite dramatic. Fader is particularly effective if your fragment contains a RecyclerView and a header, even if your RecyclerView, when it is first selected is empty, the header is still subtly animated into place.

I don’t particularly like the Slider animation but who knows what a user really likes so why not offer them a choice, using a preference setting. Of course by using a preference I have the added excuse of demonstrating how to implement a non top-level fragment.

Therefore there are several changes in the MainActivity. We have to add the interface NavigationView.IOnNavigationItemSelectedListener and then the event handler OnNavigationItemSelected(IMenuItem menuItem), which should be familiar to anyone who has used a NavigationView and finally comment out NavigationUI.SetupWithNavController(navigationView, navController) which we no longer require.

Inside OnNavigationItemSelected, we created the NavOptions and set the animations. Then a simple switch to handle all the top-level fragments. As you add new fragments to your project, you need to update your nav\_graph and update the switch here to reflect any new fragment android:ids. We then manually using navController.Navigate(menuItem.ItemId, null, navOptions) to navigate to a fragment.

The SettingsFragment is a straightforward PreferenceFragmentCompat, nothing special there. One feature to note is that because it is not a top-level fragment it automatically has an up key icon appear in the app bar, which performs the same action as the back key or a back gesture. In the OnOptionsItemSelected, we again use the navController to navigate to the correct fragment.

The OnDestinationChanged event fires as the SettingsFragment closes, so that is an ideal place to call CheckPreferenceChanges to update the value of animateFragments or any other preference setting. If the Use Screen animations preference is checked, then we immediately get the sliding fragment animations.

Another powerful feature we implemented in NavigationGraph2 is the OnBackPressedCallback. From the Google docs - This class maintains its own [enabled state](https://developer.android.com/reference/androidx/activity/OnBackPressedCallback#isEnabled()). Only when this callback is enabled will it receive callbacks to [handleOnBackPressed](https://developer.android.com/reference/androidx/activity/OnBackPressedCallback" \l "handleOnBackPressed()).

We create our own class NavFragmentOnBackPressedCallback which allows us to override OnBackPressedCallback’s HandleBackPressed() to call individual coded HandleBackPressed() methods for each fragment again optionally using NavOptions. Note the comments in the ImportFragment how it's HandleBackPressed() overcame a problem that Android 12 created. Please also note our comments in the OnResume() of each fragment and the comments in NavGragmentOnBackPressedCallback.

**NavigationGraph3 project**

This project is quite a departure from the preceding ones.

Both preceding projects are based more or less on the Android app templates provided by Xamarin. These haven’t been updated since approximately 2019 other than defaulting to use the AndroidX libraries as compared to the old Android.Support libraries and even then, as mentioned previously, we still find the Add New Item wizard for a fragment defaulting to using an Android.App.Fragment, instead of an AndroidX.Fragment.App fragment. It would appear they made minimal changes just to accommodate Android 10. They also don’t support Dark mode by default as there is no provision for themes.xml in the values and values-night folders. We can still use those templates, but we are required to do a lot of additional work, that shouldn’t be necessary if the templates had kept pace with the newer versions of Android.

If we first look at the xml files. Please use a program such as Beyond Compare to open each file from NavigationGraph2 and NavigationGraph3 side by side to observe the changes. In content\_main we have added a BottomNavigationView and therefore have had to modify the constraints of the FragmentContainerView, activity\_main has only two very subtle changes (more on that later).

In app\_bar\_main we have updated both AppBarOverlay and the PopupOverlay, to correspond with the new themes.xml.

As part of demonstrating the new themes, we have also updated the headerLayout of the NavigationView and changed its name from nav\_header\_main to just nav\_header. It is now an androidx.constraintlayout.widget.ConstraintLayout with two TextViews and a background containing an image android:background="?attr/vehicle\_image". The images are contained in the Resouces/drawable folder. The images were obtained from the web and then converted to *webp* format and added to the drawable folder.

This gets us completely away from the standard Xamarin header, so now the only resemblance to the original Xamarin template is the application icon.

In a future project, we intend to introduce a Navigator icon to complete the change but would rather wait for the release of the new AndroidX SplashScreen NuGet package which is yet to be released.

We will detail these changes below, but first I should mention that there are several links at the top of the MainActivity that you can use as a starting point for documentation.

Theming assumes branding your app. As my main app is an automotive app, I looked for ideas for different themes based on automotive images. The images then help in the design of a colour scheme. A very useful tool to help extract colours from an image is available via the Xamarin.AndroidX.Palette. The ColorPalette app is also available on my Github page. Build and run it substituting your images to get a starting point for your colour scheme.

The first major change in the code is the addition of a BottomNavigationView and the extra line of code for automatic navigation of the BottomNavigationView.

NavigationUI.SetupWithNavController(bottomNavigationView, navController);

Note the additional code in OnDestinationChange where we control the visibility of the BottomNavigationView to be only visible when the third fragment is selected. Tapping on either the Leaderboard or Register menu items opens their respective fragments. Neither of those fragments is a top-level fragment, therefore by default, they will display an up button in the app bar, this example demonstrates removing the navigation icon if required.

In my automotive app, I open fragments from a BottomNavigationView that are fully immersive, so I don’t require that extra code as the StatusBar, Toolbar and NavigationBar are all removed anyway in a fully immersive fragment. Also note that in keeping with the NavigationComponent standard navigation, both these fragments when closing navigate back to the fragment from where they were opened (e.g. SlideshowFragment), not to the HomeFragment, another back gesture or back key on the SlideshowFragment then returns to the HomeFragment.

Another change to the MainActivity is that it now inherits from BaseActivity. The BaseActivity handles a change in the ColorTheme when the app starts by checking if it has changed and if so when the color theme is changed uses SetTheme to change the chosen color theme before SetContentView is called.

The next major change that was added is IOnApplyWindowInsetsListener and its corresponding event handler OnApplyWindowInsets by using ViewCompat.SetOnApplyWindowInsetsListener(toolbar, this). This coincides with the removal of android:fitsSystemWindows="true" of the Drawerlayout and the NavigationView.

OnApplyWindowInsets introduces a couple of features, that tie together. On the one hand, we want to be able to control WindowInsets in relation to the Statusbar and on the other we want to be able to support Window.Attributes.LayoutInDisplayCutoutMode which controls whether we allow our views to enter the area of the device’s StatusBar where the device contains cutouts for the front-facing camera. This is usually important if your fragments are fully immersive. As an example, if you consider Google’s Gmail app. Open Gmail and then rotate the app, observe the area where the cutout is – it doesn’t matter if it has a corner cutout or centre cutout, you will notice that the Gmail views don’t intrude into that area. If you don’t have a device, that has a cutout you can simulate the cutout via Developer Options - Drawing/Simulate display cutout and choose from the list. The way Gmail’s view is displayed is known as “Letterboxing”. Or in code Window.Attributes.LayoutInDisplayCutoutMode = LayoutInDisplayCutoutMode.Default;

In Navigation3, if you check the preference *Devices with Notches/Cutouts allow full screen* you will notice that in Landscape mode the views do now intrude into or fill the device cutout area. Depending on what your views are displaying it is certainly possible that some data may now be hidden by the display cutout hole, so it is understandable in the Gmail instance why they would use the default value. However, if your app contains immersive fragments (as mine do) then giving the user the choice to allow the app to use the whole screen is important. Again please refer to the list of articles (the first two) in MainActivity. Interestingly Google Maps allow full-screen in Portrait mode but reverts to “Letterbox” in Landscape mode. I can’t imagine why they bother to do that, considering that a map could be easily scrolled if the cutout did hide some tiny amount of important information. Some of Google’s apps do make full use of the screen in Landscape mode for example Photos but they are careful about allowing controls and content intruding into that area.

*Theming*

The other major change in Navigation3 is the use of the themes.xml in both values and the values-night folder, which allows you to create both light and dark themes. The starting point for learning about theming is the following link <https://material.io/develop/android>.

Here I will concentrate mainly on colours. In the previous projects, the values folder colors.xml file had colours, colorPrimary, colorPrimaryDark and colorAccent and their values referenced the standard Xamarin theme colours which were referenced in Styles.xml. Those 3 colours are now known as colorPrimary (same as before), colorPrimaryVariant and colorSecondary respectively. There are now a total of 12 defined colours including android:colorBackground which is still retained. This example is missing colorSecondaryVariant as I have no use for it in any of these themes. Its default colour is a teal variation and therefore I haven’t defined it. If you decide you need colorSecondaryVariant then just add it to the theme.

There are a couple of charts (see the reference articles) showing the default colours of MDC colour attributes with light and dark baseline values that make this easier to understand. MDC means Material Design Components – you will see references to MDC throughout the documents in the references. If you are only using 3 colours as in this project, then you don’t actually have to define the other colours as they automatically default to those colours. However, by defining them now, it becomes far easier to change them when you do have that requirement.

The other change in MDC is the way we describe colours. Colour names should now describe the actual colour using literal names (describe the value), not how it is used, therefore not the colour slot they will be associated with. See colors.xml in this project for examples.

This project is not following material guidelines exactly – Google doesn’t either with some of their apps. I don’t like the look of using a desaturated colour as the colour of the StatusBar in a dark theme, so I did what Google does with Gmail and made the StatusBar black.

Please note that these dark themes work only on devices Android 10 and above.

I mentioned Styles.xml previously. Styles.xml hasn’t gone away but is now retained for you guessed it, styles.

Our only example style is AlertDialogRoundedCornersTheme in this project. Which styles our BasicDialogFragment. BasicDialogFragment inherits from AppCompatDialogFragment. When we call it, we pass two strings, the text of the title and the text of the message to display. AlertDialogRoundedCornersTheme rounds the corners of the dialog, centres the title and bolds its text.

**NOTES**

When I started this document, I said that there was a bug in BottomNavigationView. The reason I have limited the version to 2.3.5.3 was to avoid a bug that is contained in Xamarin.Google.Android.Material in version >= 1.4.0.4 The bug involves the BottomNavigationView

I’ve now worked around this issue in this project

First, we upgraded both Xamarin.AndroidX.Navigation.Fragment and Xamarin.AndroidX.Navigation.Navigation.UI to version 2.4.2.

Xamarin.AndroidX.Navigation.Navigation.UI includes Xamarin.Google.Android.Material 1.5.0.2. However Material 1.6.0 is now the latest version of Xamarin.Google.Android.Material, so after further testing we then upgraded it to 1.6.0 by adding it as a separate package.

The problem was with the upgrade to Material 1.5.0.2. The problem was not so obvious. As a first test, the fragments controlled by the BottomNavigationView displayed correctly. However, if you tried to open either the LeaderboardFragment or the RegisterFragment more than once it would fail to open on the second occasion. If you then closed the SlideshowFragment and then opened it again, then again you could open either fragment, but again it would fail when trying to open it on the second try. That was certainly different behaviour from when using 2.3.5.3 of the Navigation Nuget packages. However, the problem wasn’t with the Navigation components but with the Android.Material Nuget which was automatically upgraded to 1.5.0.2.

I believe the problem was related to the deprecation of two methods [setOnNavigationItemReselectedListener](https://developer.android.com/reference/com/google/android/material/bottomnavigation/BottomNavigationView#setOnNavigationItemReselectedListener(com.google.android.material.bottomnavigation.BottomNavigationView.OnNavigationItemReselectedListener)) and

[setOnNavigationItemReselectedListener](https://developer.android.com/reference/com/google/android/material/bottomnavigation/BottomNavigationView#setOnNavigationItemReselectedListener(com.google.android.material.bottomnavigation.BottomNavigationView.OnNavigationItemReselectedListener)), which would have been used by NavigationUI.SetupWithNavController(bottomNavigationView, navController).

These have been replaced by [NavigationBarView.setOnItemReselectedListener(OnItemReselectedListener)](https://developer.android.com/reference/com/google/android/material/navigation/NavigationBarView#setOnItemReselectedListener(com.google.android.material.navigation.NavigationBarView.OnItemReselectedListener)) and [NavigationBarView.setOnItemSelectedListener(OnItemSelectedListener)](https://developer.android.com/reference/com/google/android/material/navigation/NavigationBarView#setOnItemSelectedListener(com.google.android.material.navigation.NavigationBarView.OnItemSelectedListener))*.* My guess is that the old [setOnNavigationItemReselectedListener](https://developer.android.com/reference/com/google/android/material/bottomnavigation/BottomNavigationView#setOnNavigationItemReselectedListener(com.google.android.material.bottomnavigation.BottomNavigationView.OnNavigationItemReselectedListener)) returned a different value and the value now returned prevents the fragment from opening again. As we were using NavigationUI.SetupWithNavController(bottomNavigationView, navController) then we have avoid the issue.

Further investigation re the 2.4.0 version differences – Navigation.UI 2.4.0 that took a BottomNavigationView was updated to instead take its superclass introduced in Material 1.4.0, NavigationBarView.

So we have now removed both NavigationUI.SetupWithNavController(navigationView, navController) and NavigationUI.SetupWithNavController(bottomNavigationView, navController) which means we have to write additional code. However, if we want the simplicity of controlling animations and to have the desired behaviour of the BottomNavigationView then I guess we have to write and use the extra code.

**Proguard considerations**

Each project has already been built in Release mode, therefore a proguard.cfg is included in each project. I haven’t modified my usual proguard.cfg from my app and therefore there are probably keep rules that are not required. Having those extra keep rules, that aren’t required does not affect building release. The one important keep rule that is required is. Without it, those rules the release version of the app will crash at start-up.

-keepattributes RuntimeVisibleAnnotations

-keep class \* extends androidx.navigation.Navigator

If you wish to experiment you can remove other keep rules, but please test for each rule that is removed. There are notes in the proguard file that are worth observing especially the mapping file which is required when you upload an aab to Google Play.

Please be sure to check out the bin\debug and bin\release folders and observe the changes in apk size when using r8 and linking with SDK and User Assemblies.

**Visual Studio 2020 version used**

VS2020 17.3.0 Preview 1.0

**Update to NavigationGraph3 Project June 10th 2022**

Recently Xamarin.AndroidX.Core.SplashScreen 1.0.0-rc01 was released (still as a Preview so don’t forget to check *include prerelease* when searching for it) so we have now incorporated it in this updated version of NavigationGraph3. As you will see it is using an updated version of the Xamarin application icon. This icon is essential for demonstrating the features of the android.core.Splashscreen. Thanks to Joseph Edusei for supplying the modified icon. If you run this app without this icon, you will not get the expected results. For more information on how to create a suitable icon see <https://developer.android.com/guide/topics/ui/splash-screen>

Also, see my notes in values/themes.xml on the minimum requirements for the splash screen to work. Please note that I haven’t bothered with an animated icon, nor using any artificial delay using something like the following.

<item name="android:windowSplashScreenAnimationDuration">1000</item>

From my point of view, I want the fastest possible cold start-up times and something like the above would defeat that goal.

One other feature that could be added for devices less than Android 10 would be a splash screen based on a layer-list for example. This would then have to go in the resources/drawable folder. Perhaps even resources/drawable-v24 or v27, so it potentially can’t interfere with the new splash screen.

I’m going to attempt that in my own app and will update here if successful. When I first moved to Android 12, (prior to core.Splashscreen) this was what my existing splash screen was and it worked well, but obviously didn’t support the quick setting’s Dark Theme – On/Off as this version does.

<layer-list xmlns:android="http://schemas.android.com/apk/res/android"

android:opacity="opaque">

<item>

<color android:color="?attr/splashScreenBackgroundColor"/>

<!--<color android:color="?android:attr/colorPrimary"/>-->

</item>

<item>

<bitmap

android:src="@drawable/ic\_launcher\_round"

android:tileMode="disabled"

android:gravity="center"/>

</item>

</layer-list>

PS:

Over the next couple of days, I intend to upload an Android 12-only version, (ported from https://github.com/android/user-interface-samples) using a completely different technique, changing day/night mode within the app that doesn’t solely rely on the quick setting’s Dark Theme button.

**NavigationGraph5 Project June 27th 2022**

Please note that there is no NavigationGraph4 project. I got ahead of myself with the numbering when creating this new project.

Over the last couple of weeks, I’ve been playing around with re-designing the icon of my app as a vector or animated vector and have to admit that my skills in that area are not up to scratch and rather than becoming an icon designer, I’m going to give that job to a professional. While I wait for a new icon, I decided to utilize the old icon and make it work with Android 12, plus all the other versions of Android that my apps support.

In this example, I’ve just used the standard Xamarin icon, rather than the vector as was used in NavigationGraph3, i.e. the splash\_logo.xml.

We now directly utilise the standard Xamarin adaptive icons of the mipmap-anydpi-v26 folder. A new style file has been created in values-v31 just for Android 12 and above. The assumption is that as Android releases new versions such as Android 13 there will be new features added to the Splash Screen API that will not necessarily be backward compatible. So now the project has two styles.xml files, which for the moment are identical with regards to the SplashScreen. Obviously, devices under Android 12 use the values from the values/styles.xml.

I would therefore assume in your own apps you have the same setup of mipmap folders that this project has, therefore the splash screen styles of this project should work for any Xamarin Android project as it does for my apps with their icons. If for some reason you don’t have that folder setup, you can use Android Studio’s Image Asset tool (https://developer.android.com/studio/write/image-asset-studio) to create a set of application icons, based on your icon, to produce the icons required for your Visual Studio project. Once created in Android Studio, import them (use Add existing into each folder) into your project.

The other major change as compared to NavigationGraph3 is the SettingsFragment. A new preference has been added SystemThemeListPreference which is only available to Android 12 devices as the method used to save the selected UiNightMode is only available in Android 12 and above. Therefore the SettingsFragment on an Android device less than 12 (10,11) will have that preference disabled with instructions to use the Quick Settings Panel Theme button to change the theme from Light to Dark or vice-versa. Some devices running Android 9 can also change the theme from Light/Dark e.g. Samsung Galaxy S8, so they too respond the same way as Android 10 and 11 devices.

**Devices tested**

Pixel 6 - Android 12

Samsung Galaxy Tab S7 - Android 12 – hard to see – the device is too fast.

Samsung Galaxy S20 5G – Android 12

Pixel 4a - Android 11

Pixel 3a - Android 10

Samsung Galaxy S8 – Android 9

Nexus 5x – Android 8.1.0

Samsung Tab A – Android 7.1.1

Samsung Galaxy S6 Edge

**Testing**

Make sure the app is cleared from recents for each test to ensure a cold start so that you can see the icon. You can always uncomment Thread.Sleep() in the MainActivity’s OnCreate to get a longer view of the icon.

**Visual Studio 2022 version used**

VS2022 17.3.0 Preview 1.2

**NavigationGraph6 Project - August 30 2022**

This is an upgrade to NavigationGraph5. Two features have changed. NavigationGraph5 was accidentally upgraded to include a ViewPager2 demonstration, however, it wasn’t documented, so we will include the documentation here.

The other change is quite substantial in that with the latest release of Xamarin.AndroidX.Navigation.Fragment and Xamarin.AndroidX.Navigation.UI both from 2.4.2.1 to 2.5.1 we have several methods that have been deprecated and replaced. The methods come from AndroidX.Fragment.App as Xamarin.AndroidX.NavigationFragment 2.5.1 now depends on Xamarin.AndroidX.Fragment 1.5.1 instead of 1.4.1.2.

What this means is that we have a considerable amount of refactoring to do to get the project to build without warnings. To see the warnings temporarily upgrade both the Navigation Nugets to 2.5.1 in the *NavigationGraph5* project and then rebuild.

The deprecated methods are OnCreateOptionsMenu, OnOptionsItemSelected and OnPrepareOptionsMenu (not applicable in that project because it was never used). Also SetHasOptionsMenu (and the equivalent C# version HasOptionsMenu = bool value) is deprecated, but not picked up in this version Xamarin.AndroidX.Fragment.

I was using a technique I first came across years ago when starting out using PreferenceFragmentCompat. I found that when the SettingsFragment displayed it automatically picked up the OnCreateOptionsMenu from the MainActivity. I then learnt that you could prevent that by setting HasMenuOptions = true in OnCreate() and then in OnCreateOptionsMenu inflate the menu, but then immediately call menu.Clear() and that got rid of the menu. If you look at the SettingsFragment and some of the other fragments in NavigationGraph5 that is exactly how it is done. That seemed weird at the time and still does, but it worked and that is how I’ve always prevented an unwanted menu from displaying on a fragment.

To get around the deprecation of the methods, Android has introduced two new Interfaces IMenuProvider and IMenuHost. IMenuProvider has 2 methods OnCreateMenu(IMenu menu, MenuInflater menuInflater) and OnMenuItemSelected(IMenuItem menuItem) with very similar names and functionality to the deprecated methods.

IMenuHost has 3 versions of AddMenuProvider() and a RemoveMenuProvider() and an InValidateMenu(). For a fragment as in this example, we use the 3rd method of AddMenuProvider( IMenuProvider menuProvider, ILifecycleOwner lifecycleOwner, AndroidX.Lifecycle.Lifecycle.State state). See the HomeFragment in this project for an example. AddMenuProvider is based on LifeCycle therefore it is only applicable while this fragment is visible. Conversely, if your fragment doesn’t require a menu then you simply don’t implement the two new interfaces. See the revised SettingsFragment in NavigationGraph6 as compared to the same fragment in NavigationGraph5.

As with any Interface, the easiest way to implement the methods of the interface, especially if you are new to this particular interface is to add IMenuProvider to the class definition. Visual Studio will immediately give it a red squiggly. Right-click on it and Visual Studio will suggest that it will implement the interface for you. You then add the code you want to replace the generated System.NotImplementedException.

One point I should stress because it took me some time to figure it out. In the OnMenuItemSelected method of the HomeFragment, I originally was just returning true for the default condition. However, when the SettingsFragment closed I could no longer activate the drawerlayout NavigationView via the hamburger icon. Compare the OnMenuItemSelected method of the HomeFragment and the old OnOptionsItemSelected method (commented) in the MainActivity. NavigationUI.OnNavDestinationSelected must be called for the hamburger menu icon to keep working.

So you might ask what are the benefits of using this new interface. It certainly gets rid of the quirky procedure we went through before to ensure that a fragment that doesn’t need a menu no longer has to provide that weird code. However, the biggest benefit is that it removes code from the MainActivity that really should never have been there in the first place.

That was one of the important parts Google harped on about when they released the Navigation Component – to reduce the MainActivity from being god-like. Ironically, since I started this series of tutorials, I’ve been steadily adding code to the MainActivity to work around what I call limitations of the Navigation component, so it is nice to have a win in that regard.

What was removed from the MainActivity? I’ve left the removed code at the bottom of MainActivity (commented) just to demonstrate how much was removed.

**ViewPager2**

**Visual Studio 2022 version used**

VS2022 17.4.0 Preview 1.0

**NavigationGraph6 Project August 25th 2022**

NavigationGraph6 contains several changes. The first change is due to the deprecation of several methods in Xamarin.AndroidX.Navigation.Fragment 2.5.1 as compared to 2.4.2. The second change is the addition of a simple ViewPager2 to the Leaderboard fragment to demonstrate the differences between a ViewPager and ViewPager2.

***Differences between Xamarin.AndroidX.Navigation.Fragment 2.5.1 as compared to 2.4.2.***

The first thing you will notice if upgrading any of the previous NavigationGraph projects with Fragment 2.5.1 and NavigationUI 2.5.1 is a whole bunch of deprecation warnings on all the fragments e.g. Fragment.OnCreateOptionsMenu is obsolete: ‘deprecated’ After looking at the Google docs there should also be a warning about using HasOptionsMenu = true as SetHasOptionsMenu(Boolean) is also marked as deprecated, but that one was missed when created by the Xamarin versions. This of course breaks all our fragment code, but the solution is really useful.

Years ago I had a SettingsActivity, like most apps back then for all my user settings, then I came across the PreferenceFragmentCompat which could be used to replace the SettingsActivity, which at the time seemed an attractive way to do settings as it meant I could unload the requirement of using a specific activity just to handle settings.

During the development of PreferenceFragmentCompat one of the problems I noticed was that the fragment automatically gained a 3 dot menu, the 3 dot menu used by the MainActivity, which I didn’t need or want. In searching for a solution I came across SetHasOptionsMenu(bool) used in OnCreate and then in OnCreateOptionsMenu()inflate the menu and then immediately clear the menu items with menu.Clear(). At the time that appeared to be a weird way to get rid of a menu but it worked fine and so I’ve used that technique when I don’t need a menu on a fragment, as can be seen in all the previous NavigationGraph projects. When I did need a menu on a fragment I created a new menu for that fragment and used it. Problem solved but still a pretty weird way to do it.

New with this release there are the new IMenuProvider and IMenuHost and replacement methods OnCreateMenu and OnMenuItemSelected. The IMenuProvider interface has two methods OnCreateMenu and OnMenuItemSelected, which are very similar in functionality to the previous OnCreateMenuOptions and OnMenuOptionsSelected. The IMenuHost is where things get different. It has three AddMenuProvider methods. For a fragment, we use the third method which takes 3 parameters as can be seen in the HomeFragment. Notice how it is tied to the Lifecycle of the fragment, which means we don’t have to remember to remove it.

This certainly does improve each fragment. The fragment now only contains methods related to this fragment. See ShowSubscriptionInfoDialog as an example. Previously this method was in the MainActivity, now it is where it belongs in the fragment that called it. And for a fragment that doesn’t require a menu, we can now remove the extra code that prevented the menu from displaying.

The other advantage is that we have uncluttered the MainActivity. To demonstrate, I left the code that was removed from the MainActivity at the end of the activity as a commented-out block.

***ViewPager2 - a replacement for ViewPager***

This project also includes the replacement for ViewPager, which is ViewPager2. There a quite a few changes as compared to ViewPager and its PagerAdapter. The concept of using a list of fragments and returning the correct fragment as in return fragmentList[position] as was done with GetItem in ViewPager has been replaced, so to start since we are only dealing with Fragments we need to know about the FragmentStateAdapter.

The FragmentStateAdapter is an abstract class so we need to create our own class and implement the two abstract members, CreateFragment(int) and ItemCount. All fragment constructors are required to be parameterless, as the adapter can call the constructor at any time while paging, and during the cycle of creating and destroying the fragment. The FragmentManager is responsible for looking after the creation and destruction of the fragments as the user swipes through the pages. FragmentStateAdapter has 3 constructors, since we dealing with fragments we use the third constructor which requires the following two parameters FragmentManager fragmentManager, AndroidX.Lifecycle.Lifecycle lifecycle. Since we are free to modify this class we can pass in an additional parameter, the number of fragments or pages we want to handle, which will obviously become the ItemCount of our FragmentStateAdapter.

Once we have the FragmentStateAdapter we then need a fragment (LeaderboardPagerFragment) for the layout of ViewPager2. In this example, the layout will contain a ViewPager2 and a TabLayout. The TabLayout is not your traditional TabLayout, here we are just using it as a page indicator. That is not to say we can’t use a TabLayout as a traditional TabLayout as we can and that would provide a ViewPager that can be both tab-driven and swipable. In this example, we just have swipable pages. Admittedly, the fragment contents we display are pretty lame, as are all the other fragments contents in this series. The only reason we added a couple of extra TextViews to the three sample fragments (pages) is so that ZoomOutPageTransformer we are using is more obvious. It does work with a single-centred TextView but it needs the two extra TextViews at the top and bottom to more effectively demonstrate the effect. Try with only the single TextView and you barely see the effect.

We’ve added an OnPageChangeCallback class which has an OnPageSelected method but it is not used in this example. The same for the TabConfigurationStrategy class and its OnConfigureTab method.