# How to make a Xamarin Android video player using GStreamer

1. Everything is based on the canonical Android example by the GStreamer community [here](https://gstreamer.freedesktop.org/documentation/tutorials/android/a-complete-media-player.html?gi-language=c). My result is [here](https://github.com/kwende/GStreamerXamarin). Code for the GStreamer community examples are hosted [here](https://github.com/GStreamer/gst-docs/tree/master/examples/tutorials/android).
2. IDEs used for this example:
   1. Visual Studio 2022
   2. Android Studio Hedgehog | 2023.1.1 Patch 2
3. Go to the root of your repo (for me its D:\repos\GStreamerXamarin).
   1. Create a Xamarin folder
   2. Create a Java folder
4. Open Android Studio
   1. New Project > Phone and Tablet > No Activity
      1. Name the Project RTSPPlayer, place under directory RTSPPlayer (example: D:\repos\GStreamerXamarin\Java\RTSPPlayer).
      2. To work with the examples more easily, choose “Java” for the language, and choose “Groovy DSL” for the build.gradle language.
      3. Click Next. Wait a while for gradle sync to complete (pay attention to the dialogs on the lower right, you may get some Microsoft Defender configuration notifications, do as it says).
   2. From the menu at the top left, choose Project Files as the option. A screenshot of a computer

      Description automatically generated
   3. By default, this will build an app, or an APK. We need to make modifications to the project files (more details [here](https://developer.android.com/studio/projects/android-library#Convert)) so as to build an AAR / library instead.
      1. Expand “app”, double click the “build.gradle” file.
      2. Find “plugins” at the top and change the id from ‘com.android.application’ to ‘com.android.library’.
      3. Find the “defaultConfig” section and remove lines for “applicationId”, “versionCode”, “versionName”, and “targetSDK”.
      4. Build (Ctrl-F9). Make sure an AAR is generated. For its built here for me: D:\repos\GStreamerXamarin\Java\RTSPPlayer\app\build\outputs\aar
5. Let’s install GStreamer and the NDK. GStreamer requires a specific version of the NDK in order to operate.
   1. Go [here](https://gstreamer.freedesktop.org/download/) to download GStreamer.
      1. Scroll down to find “Android”.
      2. Take note of the current stable version and download that (1.22.10 for me).
      3. Also take note of the NDK version that’s required to run it (r21 for me).
      4. Unzip the GStreamer package somewhere. I chose D:\gstreamer-1.0-android-universal-1.22.10.
   2. Return to Android Studio.
      1. Tools > SDK Manager
      2. Select the “SDK Tools” tab
      3. Scroll down to “NDK (Side by Side)” an dmake sure the NDK required to build is checked / installed. If not, install it.
      4. Before leaving, take note of the “Android SDK Location”, the NDK will be placed under this.
      5. Close out, and open up “local.properties”.
         1. Add a new line: “nkd.dir=<Android SDK Location>/<NDK location>”.
         2. Example:“ndk.dir=C\:\\Users\\brush\\AppData\\Local\\Android\\Sdk\\ndk\\21.4.7075529”
         3. This allows Android Studio to find and use the correct NDK.
   3. Add the following environment variables to allow Android Studio’s make files (you’ll see them soon) to find the GStreamer installation directory.
      1. GSTREAMER\_ROOT\_ANDROID = D:\gstreamer-1.0-android-universal-1.22.10
      2. GSTREAMER\_SDK\_ROOT\_ANDROID = D:\gstreamer-1.0-android-universal-1.22.10
   4. Make sure everything still builds in Android Studio.
6. Return to Android Studio. Let’s configure the Android Studio project to build C/C++ files. We are going to be use the JNI to interact with the native C Gstreamer code from Java.
   1. Add a JNI folder.
      1. Right-mouse click on the “app” folder and go to New > Folder > JNI Folder.
      2. Target source set: main.
      3. Click Finish.
      4. There should be a jni folder at app/src/main/jni
   2. Add the GStreamer C Code.
      1. Add a new file under the jni folder called native.c.
      2. You should be able to straight-up copy and paste the [content of the tutorial’s C file](https://github.com/GStreamer/gst-docs/blob/master/examples/tutorials/android/android-tutorial-5/jni/tutorial-5.c) straight into native.c.
      3. You do need to change the call to “FindClass” within the method JNI\_Onload. What this does is find the Java class with the “native” methods whose implementations are to be hooked up with the C code via JNI. We’ve renamed our class from the samples, so we need to change it.   
           
         Change it to com/ocuvera/rtspplayer/Interop
   3. At this point in time Android Studio won’t know how to build this native.c file as it doesn’t know where to find headers, how to link etc. Again, the GStreamer community has a working .mk file that will help.
      1. Create a file named “Android.mk” under the jni folder.
         1. Paste the contents of [this .mk file](https://github.com/GStreamer/gst-docs/blob/master/examples/tutorials/android/android-tutorial-5/jni/Android.mk) into it. Some modifications are needed to work w/ this example, however.
            1. LOCAL\_MODULE := native
            2. LOCAL\_SRC\_FILES := native.c
      2. Create a file named ‘Application.mk”
         1. Paste the contents of [the example mk](https://github.com/GStreamer/gst-docs/blob/master/examples/tutorials/android/android-tutorial-5/jni/Application.mk) into it.
            1. For my example I removed x86, armeabi-v7a and armeabi from the APP\_ABI. This is not necessary.
      3. Now we must update the build.gradle file to introduce the mk files we generated into the build cycle. We will use [this build.gradle](https://github.com/GStreamer/gst-docs/blob/master/examples/tutorials/android/android-tutorial-5/build.gradle) as the example.
         1. Open the build.gradle file under “app.
         2. Note the “externalNativeBuild” section under “defaultConfig” under “android” [in the sample file](https://github.com/GStreamer/gst-docs/blob/master/examples/tutorials/android/android-tutorial-5/build.gradle) mentioned previously. We must copy and paste this section into the same spot within our build.gradle file.
            1. Under the “arguments” subheading, modify the value of “NDK\_APPLICATION\_MK” to be: “src/main/jni/Application.mk”. This adds the “Application.mk” file we added to the build cycle.
            2. Change “targets” to be equal to “native”
            3. Modify the abiFilters to only use “arm64-v8a” and “x86\_64”. This enables emulator support (x64).
         3. See how in the example there is an “externalNativeBuild” section. This needs to be copied to the exact spot in our build.gradle file as well.
            1. Modify path to be “path 'src/main/jni/Android.mk'“
         4. While we’re in the build.gradle file, add the following just after “compileOptions”: “ndkVersion '21.4.7075529'” (if that’s your NDK version, that is).
            1. This forces the entire build.gradle process to get on the same NDK version. There might be a better way, but this works for me.
      4. Now let’s remove some content we don’t need as we aren’t building an app, but a library:
         1. Remove the entire “res” folder (all of the resources that would have been associated with the app),
         2. Open the AndroidManifest.xml file at app > src > main. Remove the entire Application node.
      5. Attempt to build. Everything should compile fine.
         1. There should be an AAR located here (or wherever it is for your system): D:\repos\GStreamerXamarin\Java\RTSPPlayer\app\build\outputs\aar
         2. Rename the AAR to have a .zip extension
         3. Extract it.
         4. Open the folder.
         5. Navigate to jni\arm64-v8a, and you should see the C++ shared library, the libnative.so file (the C code we added) and libgstreamer\_andrtoid.so, which is a single library made of all the standard GStreamer plugins + stock bootstrap code provided by the GStreamer community.
7. Now it’s time to bind the C code to Java code through the JNI. Doing so will enable C# to invoke the Java through a binding library later, which by virtue of that binding allows C# to indirectly call the GStreamer native code in our final Xamarin app.
   1. The very first code we’ll add is “GStreamer.java”, which is stub code provided by the GStreamer community that allows us to spin up GStreamer (using the JNI) through one function call. We will use it later.
      1. Recall GStreamer is installed (for me) at D:\gstreamer-1.0-android-universal-1.22.10.
      2. Create a new package under the “java” folder in Android Studio, name it org.freedesktop.gstreamer.
      3. Navigate to D:\gstreamer-1.0-android-universal-1.22.10\arm64\share\gst-android\ndk-build and snag “Gstreamer.java” and copy it into this package directory.
      4. Make sure the project still builds.
   2. The second code we’ll add is the main class file that uses JNI to invoke the native.c code we added. This Java code will eventually be called by the C# code in Xamarin.
      1. Verify the existing of the package: com.ocuvera.rtsplibrary.
      2. Create a new file “Interop.java”.
      3. Add the following fields:   
         A screenshot of a computer program

         Description automatically generated
         1. It’s important to note that in Java there exists the JNI, a system through which calls to C can be made via Java.   
              
            Look at the “native.c” file we added earlier and scan for the field “native\_methods”. Within the method “JNI\_Onload” in the same file, we see this array being used to register the native methods. The java runtime will look within the packages to find these methods in Java (marked with native) and hook up the corresponding C function to this Java method, thereby defining it. This is how C# can call Java, which in turn can call C.   
              
            NOTE: *We will see, shortly, the importance of “native\_custom\_data” and “is\_initialized” as those aren’t used in native.c.*
      4. Each one of these native methods and fields is marked private, so dutifully go through and create a corresponding public method or accessor to invoke/retrieve each of them (with the exception of “native\_custom\_data” as that stays private to the class). This will be the public method invoked by C#, eventually. Example:   
         A screen shot of a computer program

         Description automatically generated
   3. The last piece of code we’ll add is that of the GStreamer-specific Surface view. The surface view is a component in the Android ecosystem which you can draw upon, and this is the main surface on which the video feed will be rendered by GStreamer.   
        
      There is already code built into the GStreamer runtime for drawing onto this surface, you simply have to hook it up.
      1. Add the [following code](https://github.com/GStreamer/gst-docs/blob/master/examples/tutorials/android/android-tutorial-5/src/org/freedesktop/gstreamer/tutorials/tutorial_5/GStreamerSurfaceView.java) to the com.ocuvera.rtspplayer package.
      2. Make sure the package at the top is com.ocuvera.rtspplayer.
      3. You will eventually instantiate this within the Xamarin code as a control.
      4. Make sure everything builds.
      5. Finally add the following private methods to your class:   
         A screen shot of a computer program

         Description automatically generatedIf you look at the method “gst\_native\_class\_init” which is tied to nativeClassInit, the method and property Ids for these are maintained so that they may be accessed and invoked from the C code (so, the reversal of what we’ve been doing so far with the native methods).   
           
         NOTE: *we won’t use much of these, but in advanced scenarios these are important to hook up and use. And if we don’t even in the test example, the native code as-is will fail out.*
      6. Make sure everything builds.
8. It’s now time to consume all of this from a Xamarin project.
   1. Open Visual Studio 2022.
   2. Create New Project > Mobile App (Xamarin.Forms).
      1. Location: D:\repos\GStreamerXamarin\Xamarin
      2. Project Name: XamarinPlayerApp
      3. Place solution and project in the same directory
      4. Create > Uncheck iOS (only Android)
      5. Create
   3. Create a new binding project
      1. Right mouse click XamarinPlayerApp solution > Add > New Project > Android Bindings Library (Xamarin)
      2. Project Name: XamarinPlayerAppBindingLibrary
   4. Move the ARR that we’ve built to the binding library
      1. Copy "D:\repos\GStreamerXamarin\Java\RTSPPlayer\app\build\outputs\aar\app-debug.aar" to D:\repos\GStreamerXamarin\Xamarin\XamarinPlayerAppBindingLibrary\Jars
      2. Include it in the project via Visual Studio
      3. Set the “Build Action” (right-mouse click, properties on the file) to “AndroidLibrary.
      4. Rebuild the binding project. There should be a ~26 megabyte DLL at D:\repos\GStreamerXamarin\Xamarin\XamarinPlayerAppBindingLibrary\bin\Debug.
      5. Go to the XamarinPlayerApp.Android project and add the binding library as a project reference.
      6. Right-mouse click on the XamarinPlayerApp.Android project and select Properties.
         1. Select Android Manifest
         2. We are targeting Oreo (v 26) or higher in our Android Studio project, so set “Minimum Android version” to “Android 8.0 (API Level 26 – Oreo)
      7. Rebuild the entire solution.
   5. Build a View and ViewRenderer
      1. A ViewRenderer is a Xamarin concept where you can late-bind a control to be rendered. This enables different controls to be rendered depending on whether you’re operating under iOS or Android, for example.
      2. The view renderer’s rendering pipeline also offers us a good way to initialize and setup our control from C#.
         1. Right-mouse click on XamarinPlayerApp > Add > Class. Name the class “CameraView”.
         2. Make sure the class is public and inherits from View.
         3. This is a control you can use in Xamarin.
      3. Right-mouse click on XamarinPlayerApp.Android > Add > Class. Name the class “AndroidPlayerRender”.
         1. Decorate the namespace of with [assembly: ExportRenderer(typeof(CameraPreview), typeof(AndroidPlayerRenderer))]. This tells Xamarin to render THIS when it encounters a “CameraView” View in Xamarin.
         2. Inherit the AndroidPlayerRenderer from ViewRenderer<CameraView, Com.Ocuvera.Rtspplayer.GStreamerSurfaceView>, ISurfaceHolderCallback
         3. When all is said and done, the class should look like this: A screenshot of a computer program

            Description automatically generated
         4. Note the ISurfaceHolderCallback. Every surface view in Android has a “surface view holder”, which is responsible for maintaining the current surface view. Apparently these surface views are created and destroyed when the app rotates, etc. and so if you want to access the most recent surface view, you do so by getting notifications of creation through the surface view holder. You must be of type “ISurfaceHolderCallback” to receive these notifications.
         5. Override OnElementChanged and implement it as such: A screen shot of a computer program

            Description automatically generated
         6. Implement SurfaceCreated callback as such: A screen shot of a computer

            Description automatically generated
         7. Now whenever the element is rendered we get a chance to initialize the class and create an instance, and then hook up for the various events to start playing.
   6. Load the requisite SO files. We need to load the gstreamer library as well as the native JNI interface we compiled. Go to “MainActivity.cs” within the XamarinPlayerApp.Android project.
      1. Right before “LoadApplication” in “OnCreate”, invoke the following:   
           
         A black background with white text

         Description automatically generated
      2. When those libraries are loaded, their JNI hooks will be configured by the Java runtime.
   7. Take a Xamarin page and add the CameraView. Now, when the camera view gets loaded, the ViewRenderer will get instantiated, the GStreamerSurfaceView gets instantiated, and when the surface is created, we get a callback from the holder, grab the surface, and then pass it down to GStreamer.   
        
      A screen shot of a computer screen

      Description automatically generated

A room with a table and chairs

Description automatically generated