# Xamarin (v2.3.x)

## Introduction

Xamarin is unique by offering a single language – C#, class library, and runtime that works across all three mobile platforms of iOS, Android, and Windows Phone (Windows Phone's native language is already C#), while still compiling native (non-interpreted) applications that are performant enough even for demanding games.

Each of these platforms has a different feature set and each varies in its ability to write native applications – that is, applications that compile down to native code and that interop fluently with the underlying Java subsystem. For example, some platforms only allow apps to be built in HTML and JavaScript, whereas some are very low-level and only allow C/C++ code. Some platforms don't even utilize the native control toolkit.

Xamarin is unique in that it combines all of the power of the native platforms and adds a number of powerful features of its own, including:

1. **Complete Binding for the underlying SDKs** – Xamarin contains bindings for nearly the entire underlying platform SDKs in both iOS and Android. Additionally, these bindings are strongly-typed, which means that they're easy to navigate and use, and provide robust compile-time type checking and during development. This leads to fewer runtime errors and higher quality applications.
2. **Objective-C, Java, C, and C++ Interop** – Xamarin provides facilities for directly invoking Objective-C, Java, C, and C++ libraries, giving you the power to use a wide array of 3rd party code that has already been created. This lets you take advantage of existing iOS and Android libraries written in Objective-C, Java or C/C++. Additionally, Xamarin offers binding projects that allow you to easily bind native Objective-C and Java libraries using a declarative syntax.
3. **Modern Language Constructs** – Xamarin applications are written in C#, a modern language that includes significant improvements over Objective-C and Java such as *Dynamic Language Features* , *Functional Constructs* such as *Lambdas* , *LINQ* , *Parallel Programming* features, sophisticated *Generics* , and more.
4. **Amazing Base Class Library (BCL)** – Xamarin applications use the .NET BCL, a massive collection of classes that have comprehensive and streamlined features such as powerful XML, Database, Serialization, IO, String, and Networking support, just to name a few. Additionally, existing C# code can be compiled for use in an applications, which provides access to thousands upon thousands of libraries that will let you do things that aren't already covered in the BCL.
5. **Modern Integrated Development Environment (IDE)** – Xamarin uses Xamarin Studio on Mac OS X and Visual Studio on Windows. These are both modern IDE's that include features such as code auto completion, a sophisticated Project and Solution management system, a comprehensive project template library, integrated source control, and many others.
6. **Mobile Cross Platform Support** – Xamarin offers sophisticated cross-platform support for the three major mobile platforms of iOS, Android, and Windows Phone. Applications can be written to share up to 90% of their code, and our Xamarin.Mobile

library offers a unified API to access common resources across all three platforms. This can significantly reduce both development costs and time to market for mobile developers that target the three most popular mobile platforms.

## How Does Xamarin Work?

Xamarin offers two commercial products: Xamarin.iOS and Xamarin.Android. They're both built on top of *Mono*, an open-source version of the .NET Framework based on the published .NET ECMA standards. Mono has been around almost as long as the .NET framework itself, and runs on nearly every imaginable platform including Linux, Unix, FreeBSD, and Mac OS X.

On iOS, Xamarin's *Ahead-of-Time* (*AOT*) Compiler compiles Xamarin.iOS applications directly to native ARM assembly code. On Android, Xamarin's compiler compiles down to *Intermediate Language* (*IL*), which is then *Just-in-Time* (*JIT*) compiled to native assembly when the application launches.

In both cases, Xamarin applications utilize a runtime that automatically handles things such as memory allocation, garbage collection, underlying platform interop, etc.

## Xamarin.Forms

Xamarin.Forms is a framework that allows developers to rapidly create cross platform user interfaces. It provides it's own abstraction for the user interface that will be rendered using native controls on iOS, Android, Windows, or Windows Phone. This means that applications can share a large portion of their user interface code and still retain the native look and feel of the target platform.

Xamarin.Forms allows for rapid prototyping of applications that can evolve over time to complex applications. Because Xamarin.Forms applications are native applications, they don't have the limitations of other toolkits such as browser sandboxing, limited APIs, or poor performance. Applications written using Xamarin.Forms are able to utilize any of the API's or features of the underlying platform, such as (but not limited to) CoreMotion, PassKit, and StoreKit on iOS; NFC and Google Play Services on Android; and Tiles on Windows. In addition, it's possible to create applications that will have parts of their user interface created with Xamarin.Forms while other parts are created using the native UI toolkit.

Xamarin.Forms applications are architected in the same way as traditional cross-platform applications. The most common approach is to use Portable Libraries or Shared Projects to house the shared code, and create platform specific applications that will consume the shared code.

There are two techniques to create user interfaces in Xamarin.Forms. The first technique is to create UIs entirely with C# source code. The second technique is to use *Extensible Application Markup Language* (XAML), a declarative markup language that is used to describe user interfaces. For more information about XAML, see XAML Basics.

## Instalation

To start developing Xamarin application first you need to install Visual Studio or Xamarin Studio and make sure to include all necessary Xamarin components. In our examples we will use Visual Studio to show you how to develop Xamarin application and use our SDK. Navigate to this link to read step by step how to download and install Visual Studio for Xamarin applications.

## Changelog

### version 1.0.4

- Update README.md

### version 1.0.3

- Bug fixes
- Custom renderer events changed

### version 1.0.2

- Custom renderer implementation changed.
- One Reader Device object is used for every page where scanning will be implemented

### version 1.0.1

- Update to cmbSDK v. 2.0.1
- Handling multicode

## Xamarin.Android

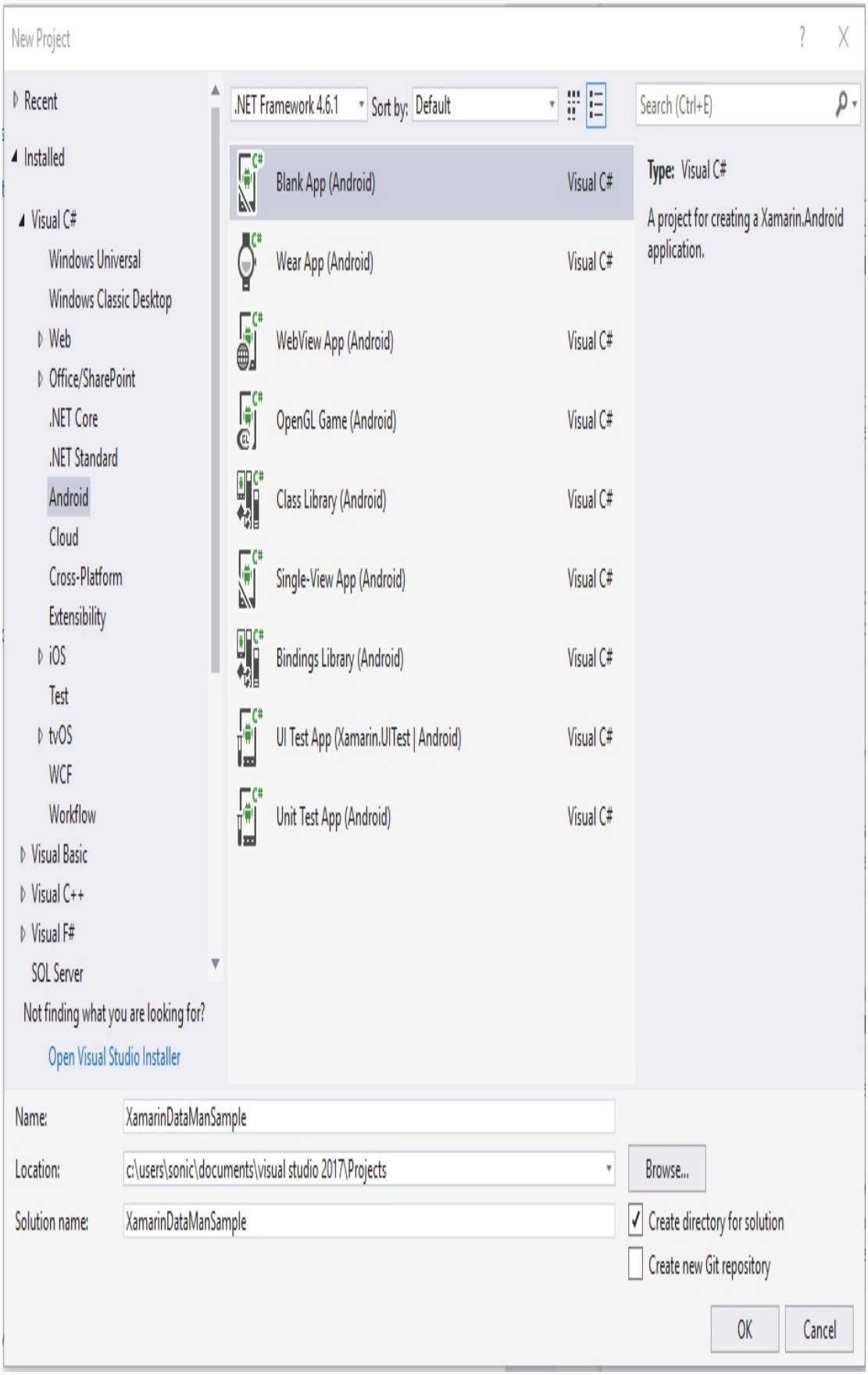
## Getting Started

In the following sections we will explain how our sample app is developed step by step.

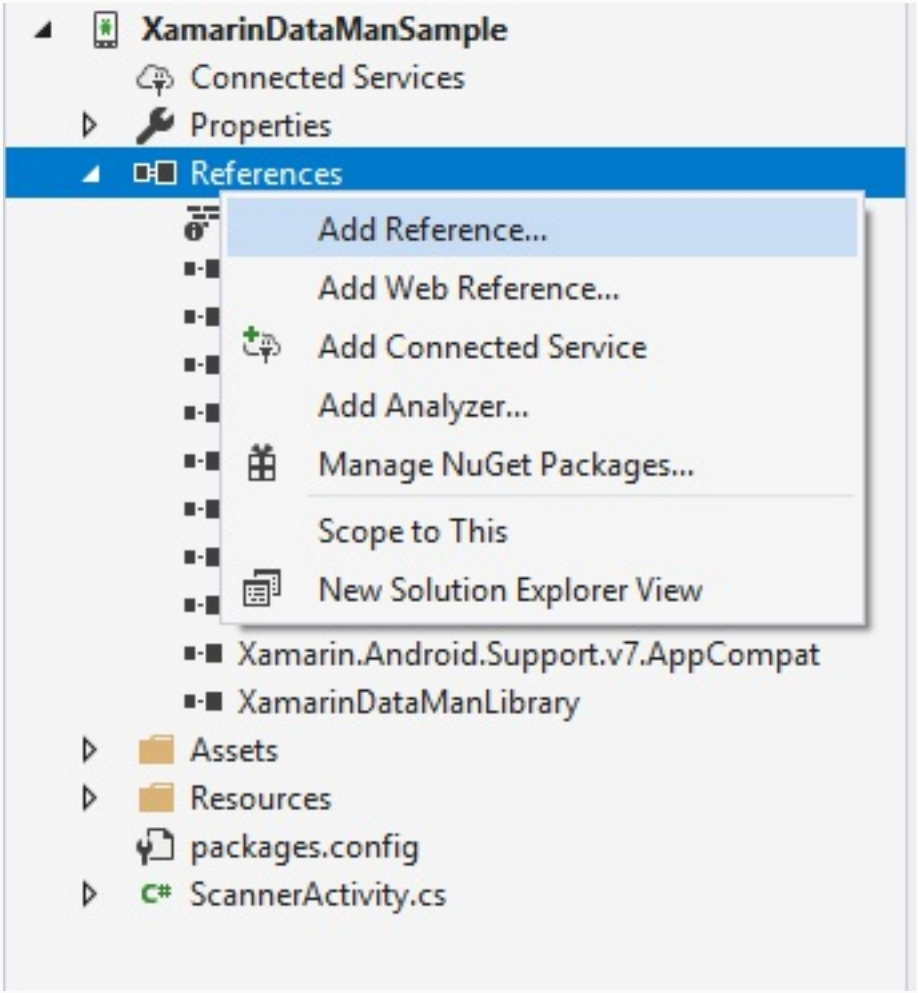
Open Visual Studio and follow these steps:

1. Go to **File -> New -> Project.**

2. Create **Blank App (Android).**



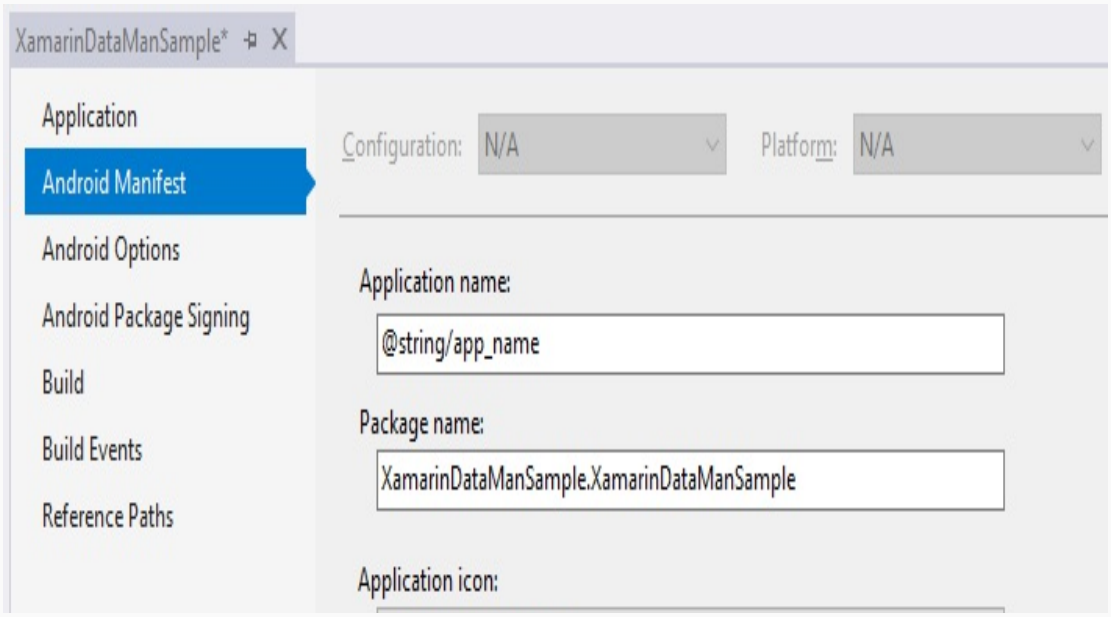

When your new project is loaded add a reference to the XamarinDataManLibrary.dll file.

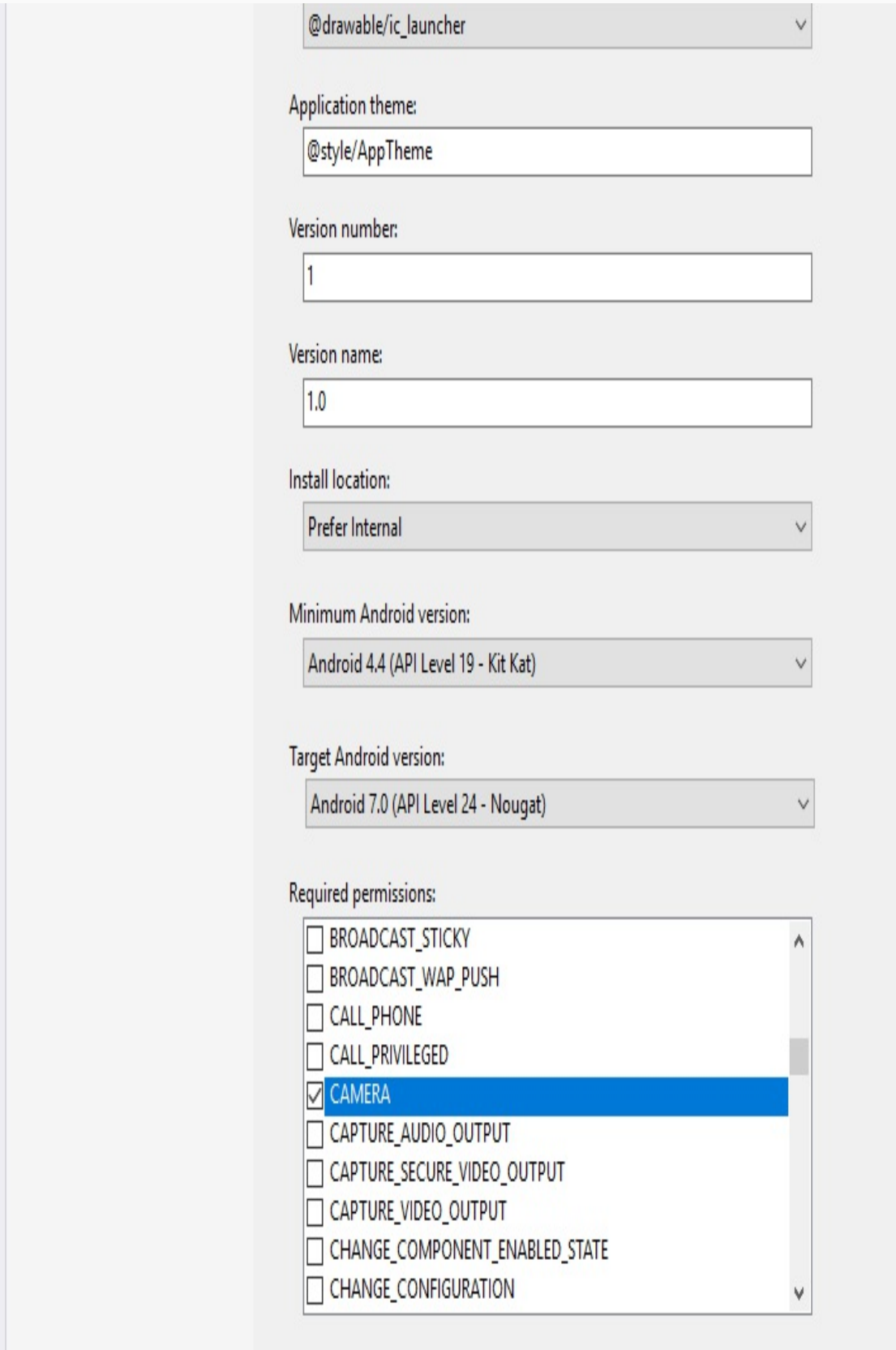



After creating a blank application, create all resources you will use (icons, images, styles, layouts, etc.). You can copy them from our sample.

Next right click on your project file, then click Properties and go to Android Manifest section.

Setup your Manifest file (app name, app icon, minimum and maximum android versions), make sure to enable Camera permission for this application.






## Scanner Activity

After we've set some necessary properties, we can create the **ScannerActivity** which

will be the **MainLauncher** for this application, and inherit some Interfaces.

```
[Activity(Label = "@string/app_name", MainLauncher = true, Icon = "@drawable/ic_launcher", ScreenOri
entation = ScreenOrientation.Portrait)]
    public class ScannerActivity : Activity, IOnConnectionCompletedListener, IReaderDeviceListener,
Android.Support.V4.App.ActivityCompat.IOnRequestPermissionsResultCallback
    {
        private static int REQUEST_PERMISSION_CODE = 12322;

        private ListView listViewResult;
        private TextView tvConnectionStatus;
        private Button btnScan;
        private RelativeLayout rlPreviewContainer;
        private ImageView ivPreview;

        private List<ReadResult> resultList;
        private JavaList<IDictionary<string, object>> resultListData;
        private ResultListViewAdapter resultListAdapter;
        public static int listViewResultSelectedItem = -1;

        private bool isScanning = false;


        private static ReaderDevice readerDevice;

        private enum DeviceType { MX, PHONE_CAMERA }

        private static bool isDevicePicked = false;
        private static DeviceType param_deviceType = DeviceType.PHONE_CAMERA;

        private static bool dialogAppeared = false;

        private static string selectedDevice = "";

        private bool listeningForUSB = false;

        protected override void OnCreate(Bundle savedInstanceState)
        {
            base.OnCreate(savedInstanceState);

            SetContentView(Resource.Layout.activity_scanner);

         ......
```

Here we will define some UI elements and variables that will be used later in the activity:

**tvConnectionStatus** - TextView an UI element for current connection status;

**listViewResult** - ListView an UI element for results that will be read;

**btnScan** - Button an UI element that will trigger ***readerDevice.StartScanning*** or ***readerDevice.StopScanning***;

**rlPreviewContainer** - ViewGroup (RelativeLayout) container for camera preview;

**ivPreview** - ImageView control with matching size as its parent for showing the last frame of a preview or scanning session;

**ReaderDevice** - cmbSDK object that will present MX Device or Phone Camera depends of our configuration;

## Configure ReaderDevice

Every time the activity starts, we are calling the **initDevice** where we configure the reader device object.

If we want to use an MX Device for scanning we can use:

```
readerDevice = GetMXDevice(this);
if (!listeningForUSB)
{
    readerDevice.StartAvailabilityListening();
    listeningForUSB = true;
}
```

The availability of the MX Device can change when the device turns **ON** or **OFF**, or if the USB cable gets connected or **disconnected**, and this is handled by the **IReaderDeviceListener** interface.

If we want to configure the reader device as a Phone Camera we can use:

```
readerDevice = GetPhoneCameraDevice(this, CameraMode.NoAimer, PreviewOption.Defaults, rlPreviewConta
iner);
```

The CameraMode parameter is of type **CameraMode** (defined in **CameraMode.java**) and it accepts one of the following values:

- **NO_AIMER**: Initializes the reader to use a live-stream preview (on the mobile device screen ), so that the user can position the barcode within the camera's field of view for detection and decoding. Use this mode when the mobile device does not have an aiming accessory.
- **PASSIVE_AIMER**: Initializes the reader to use a passive aimer, an accessory attached to the mobile device or mobile device case that uses the built-in LED flash of the mobile device as a light source for projecting an aiming pattern. In this mode no live-stream preview is presented on the device screen, since an aiming pattern will be projected.

- **FRONT_CAMERA**: Initializes the reader to use the mobile front facing camera of the device, if available (not all mobile devices have a front camera). This is an unusual but possible configuration. Most front facing cameras do not have auto focus and illumination, and provide significantly lower resolution images. This option should be used with care. In this mode illumination is not available.

All of the above modes provide the following default settings for the reader:

- The rear camera is used.
- The zoom feature is available and a button to control it is visible on the live-stream preview (if displayed).
- The simulated hardware trigger is disabled.
- When the ***startScanning()*** is called, the decoding process is started (See PreviewOption.PAUSED below for more details).

Based on the selected mode, the following additional options and behaviors are set:

- **NO_AIMER** (NoAimer)
- The live-stream preview is displayed when the ***startScanning()*** method is called.
- Illumination is available, and a button to control it, is visible on the live-stream preview.
- If commands are sent to the reader for aimer control, they will be ignored.

- **PASSIVE_AIMER** (Passive Aimer)
- The live-stream preview will not be displayed when the ***startScanning()*** method is called.
- Illumination is not available, and the live-stream preview will not have an illumination button.
- If commands are sent to the reader for illumination control, they will be ignored, since it is assumed in this mode that the built-in LED of the mobile device is being used as the aimer.

- **FRONT_CAMERA** (FrontCamera)
- The live-stream preview is displayed when the ***startScanning()*** method is called.
- The front camera is used.
- Illumination is not available and the live-stream preview will not have an illumination button.
- If commands are sent to the reader for aimer or illumination control, they will be ignored.

The ***PreviewOption*** parameter is a type of ***PreviewOption*** (defined in **PreviewOption.java**), and is used to change the reader's default values or override defaults derived from the selected *CameraMode*. Multiple options can be specified by OR-ing them when passing the parameter. The available options are:

- **DEFAULTS**: Option to accept all defaults set by the **CameraMode**.
- **NO_ZOOM_BUTTON**: Option to hide the zoom button on the live-stream preview, preventing a user from adjusting the mobile device camera's zoom.
- **NO_ILLUMINATION_BUTTON**: Option to hide the illumination button on the live-stream preview, preventing a user from toggling the illumination.
- **HARDWARE_TRIGGER**: Option to enable a simulated hardware trigger (the volume down button ) for starting scanning on the mobile device. This button only starts scanning when pressed. It does not need to be held like a more traditional purpose-built scanner's trigger. Pressing the button a second time does not stop the scanning process.
- **PAUSED**: If using a live-stream preview, when this option is set, the preview will be displayed when the ***startScanning()*** method is called, but the reader will not start decoding (i.e. looking for barcodes) until the user presses the on-screen scanning button to actually start the scanning process.
- **ALWAYS_SHOW**: Option to force a live-stream preview to be displayed, even if an aiming mode has been selected (e.g. CameraMode == PASSIVE_AIMER).

## Connecting to Device

After configuring the **ReaderDevice** we need to connect to the device.

Before we make a connection the **ReaderDeviceListener** object is set in order to receive events:

```
readerDevice.SetReaderDeviceListener(this);
```

Additionally, you can enable sending the last triggered image and SVG from the reader by:

```
readerDevice.EnableImage(true);
readerDevice.EnableImageGraphics(true);
```

Then we can connect with:

```
readerDevice.Connect(this);
```

Events that will be invoked are:

```
public void OnConnectionStateChanged(ReaderDevice reader)
public void OnConnectionCompleted(ReaderDevice reader, Throwable error)
```

If there is an error while trying to connect, the error will be thrown as a parameter in the **OnConnectionCompleted** method, otherwise, if no error occurs, the error parameter will be **null**.

If the connection is successful, the statement **reader.ConnectionState == ConnectionState.Connected** will be true.

There are couple of API methods for changing some public properties for configuring the connected device and you should invoke them when the **ConnectionState** is connected.

For example if Mobile Camera is used as a **ReaderDevice** there are **no symbologies enabled by default**. You must enable the symbologies that you want to use with the **SetSymbologyEnabled** API method:

```
readerDevice.SetSymbologyEnabled(Symbology.C128, true, null);
readerDevice.SetSymbologyEnabled(Symbology.Datamatrix, true, null);
readerDevice.SetSymbologyEnabled(Symbology.UpcEan, true, null);
readerDevice.SetSymbologyEnabled(Symbology.Qr, true, null);
```

You can do the same directly by sending a command to the connected device with:

```
readerDevice.DataManSystem.SendCommand("SET SYMBOL.MICROPDF417 ON");
```

## Scanning Barcodes

With a properly configured reader, you are now ready to scan barcodes. This can be done by calling the ***startScanning*** method from your **ReaderDevice** object.

What happens next is based on the type of **ReaderDevice** and how it has been configured, but in general:

- If using an MX Device, the user can press a trigger button on the device to turn the scanner on and read a barcode;

- If using the camera reader, the **cmbSDK** starts the camera, displays the configured live-stream preview, and begins analyzing the frames from the video stream, looking for a configured barcode symbology;

Scanning stops under one of the following conditions:

- The reader found and decoded a barcode;
- The user released the trigger or pressed the stop button on the live-stream preview screen;
- The camera reader timed out without finding a barcode;
- The application itself calls the ***stopScanning()*** method.

When a barcode is decoded successfully (the first case), you will receive a **ReadResults** iterable result collection object in the **ReaderDevice** listener method.

If your MX Device is configured to work with multi code scanning, you can access all the scanned results from the **results.SubResults** property which is an array that contains **ReaderResult** objects and it will be **null** if single code scanning is used.

## Example

```
public void OnReadResultReceived(ReaderDevice reader, ReadResults results)
        {

            listViewResultSelectedItem = -1;
            resultList.Clear();
            resultListData.Clear();
            ivPreview.SetImageBitmap(null);

            if (results.SubResults != null && results.SubResults.Count > 0)
            {
                foreach (ReadResult subResult in results.SubResults)
                {
                    if (subResult.IsGoodRead)
                    {
                        resultList.Add(subResult);

                        JavaDictionary<string, object> item = new JavaDictionary<string, object>();
                        item.Add("resultText", subResult.ReadString);

                        Symbology sym = subResult.Symbology;
                        if (sym != null)
                            item.Add("resultType", subResult.Symbology.Name);
                        else
                            item.Add("resultType", "UNKNOWN SYMBOLOGY");

                        resultListData.Add(item);
```

```
                        listViewResultSelectedItem = resultListData.Count - 1;
                    }
                    else
                    {
                        resultList.Add(subResult);

                        JavaDictionary<string, object> item = new JavaDictionary<string, object>();
                        item.Add("resultText", "NO READ");
                        item.Add("resultType", "");

                        resultListData.Add(item);
                        listViewResultSelectedItem = resultListData.Count - 1;
                    }

                    if (subResult.Image != null)
                    {
                        ivPreview.SetImageBitmap(renderSvg(subResult.ImageGraphics, subResult.Image)
);
                    }
                    else
                    {
                        if (subResult.ImageGraphics != null)
                        {
                            ivPreview.SetImageBitmap(renderSvg(subResult.ImageGraphics, ivPreview.Wi
dth, ivPreview.Height));
                        }
                        else
                            ivPreview.SetImageBitmap(null);
                    }
                }
            }
            else if (results.Count > 0)
            {
                ReadResult result = results.GetResultAt(0);

                if (result.IsGoodRead)
                {
                    resultList.Add(result);

                    JavaDictionary<string, object> item = new JavaDictionary<string, object>();
                    item.Add("resultText", result.ReadString);

                    Symbology sym = result.Symbology;
                    if (sym != null)
                        item.Add("resultType", result.Symbology.Name);
                    else
                        item.Add("resultType", "UNKNOWN SYMBOLOGY");

                    resultListData.Add(item);
                    listViewResultSelectedItem = resultListData.Count - 1;
                }
                else
                {
                    resultList.Add(result);

                    JavaDictionary<string, object> item = new JavaDictionary<string, object>();
                    item.Add("resultText", "NO READ");
                    item.Add("resultType", "");

                    resultListData.Add(item);
                    listViewResultSelectedItem = resultListData.Count - 1;
```

```
                }

                if (result.Image != null)
                {
                    ivPreview.SetImageBitmap(renderSvg(result.ImageGraphics, result.Image));
                }
                else
                {
                    if (result.ImageGraphics != null)
                    {
                        ivPreview.SetImageBitmap(renderSvg(result.ImageGraphics, ivPreview.Width, iv
Preview.Height));
                    }
                    else
                        ivPreview.SetImageBitmap(null);
                }
            }

            isScanning = false;
            btnScan.Text = "START SCANNING";
            resultListAdapter.NotifyDataSetChanged();
        }
```

**result.Image** is the last frame from the scanning process and it will be displayed in the **ivPreview ImageView,** and **result.ImageGraphics** is SVG image that locate barcode on image.

## Disconnecting from Device

In the **ScannerActivity** we override the **OnPause** and the **OnStop** events so we can do the **Disconnect** and the ***StopAvailabilityListening*** to release all connection when we navigate from or destroy that activity.

```
protected override void OnPause()
{
      base.OnPause();

      if (readerDevice != null)
      {
          readerDevice.Disconnect();
      }
}


protected override void OnStop()
{
      if (readerDevice != null)
        try
        {
             readerDevice.StopAvailabilityListening();
        }
        catch (System.Exception e) { }
```

```
        listeningForUSB = false;

        base.OnStop();
}
```

Keep in mind there might be cases when a device disconnects due to low battery condition or manual cable disconnection.

## Licensing the SDK

If you plan to use the **cmbSDK** to do mobile scanning with a smartphone or a tablet (without the MX mobile terminal), the SDK requires the installation of a license key. Without a license key, the SDK will still operate, although scanned results will be blurred (the SDK will randomly replace characters in the scan result with an asterisk character).

Contact your Cognex Sales Representative for information on how to obtain a license key including trial licenses which can be used for 30 days to evaluate the SDK.

After obtaining your license key there are two ways to add your license key in an application.

The first one is to add it as a meta tag in application tag in your manifest file:

```
<application android:label="XamarinDataManSample">
        .....

  <meta-data android:name="MX_MOBILE_LICENSE" android:value="YOUR_MX_MOBILE_LICENSE"/>
</application>
```

Second way is to implement the activation directly from code. When you create your readerDevice set license key as input parameter in constructor:

```
...
readerDevice = GetPhoneCameraDevice(this, CameraMode.NoAimer, PreviewOption.Defaults, rlPreviewConta
iner, "YOUR_MX_MOBILE_LICENSE");
```

## Xamarin.iOS

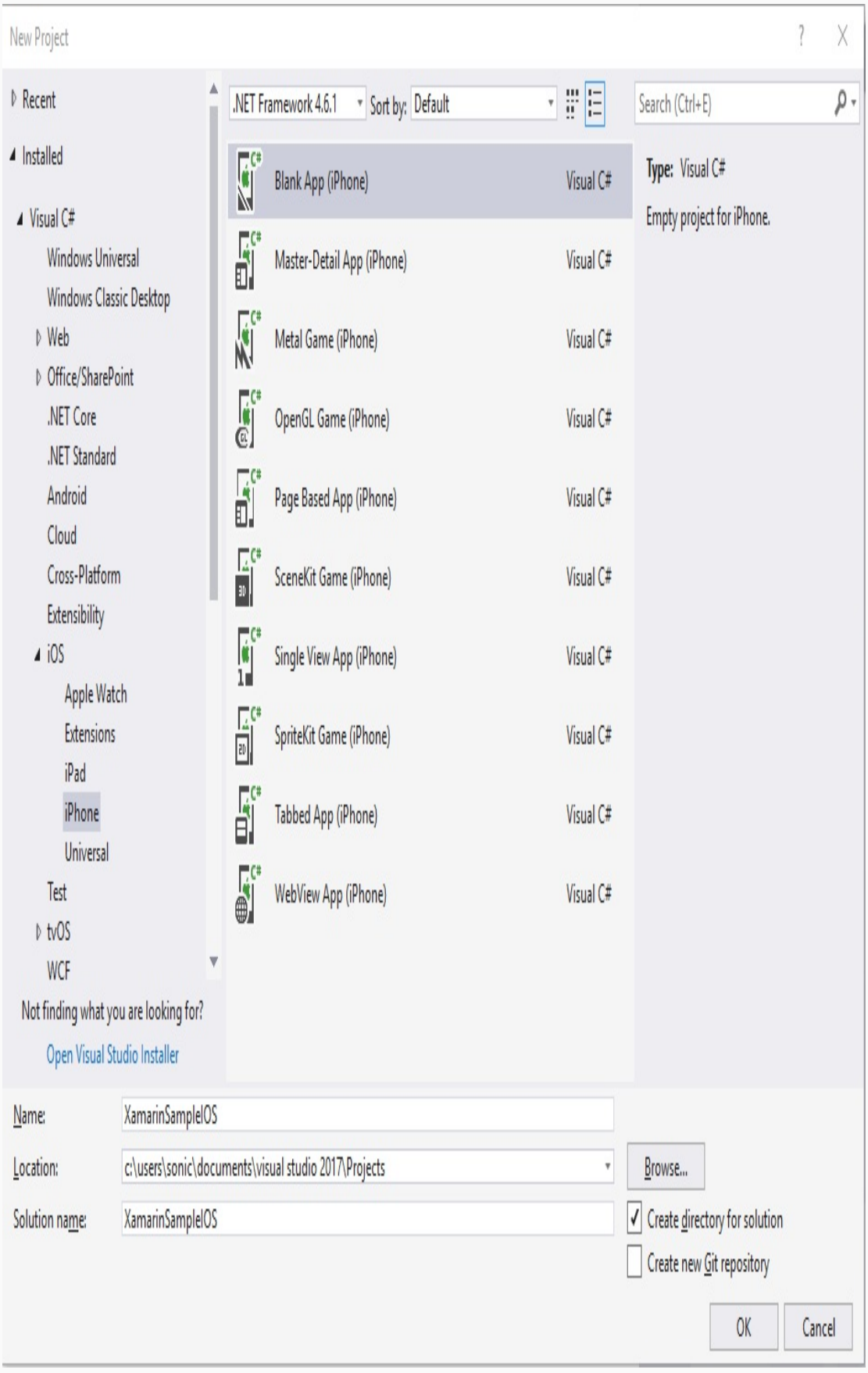
## Getting Started

In the following sections we will explain how our sample app is developed step by step.

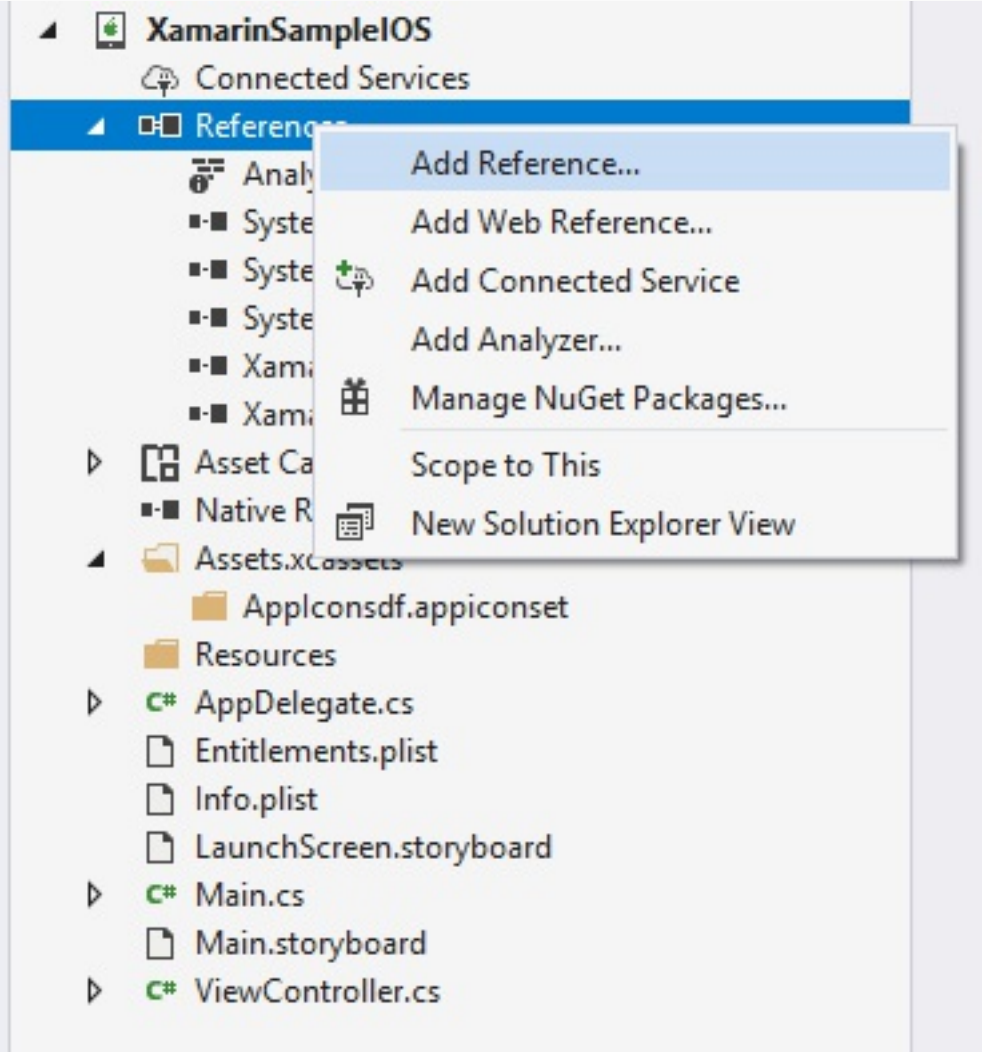
Open Visual Studio and follow this steps:

1. Go to **File -> New -> Project.**

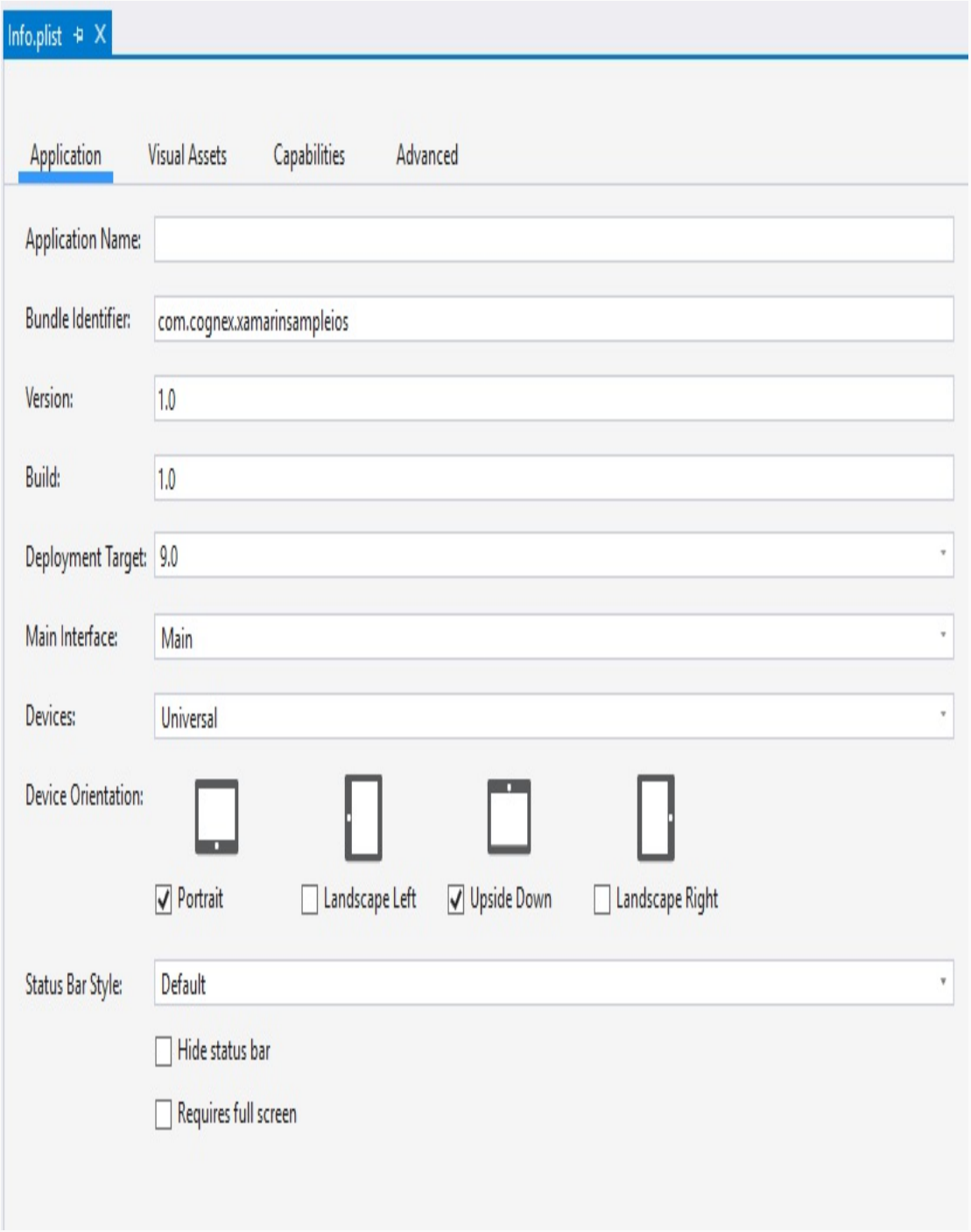
2. Create **Blank App (iPhone).**




When your new project is loaded add reference to **XamarinDataManLibrary.dll** file.




Next open your **Info.plist** file and set some project properties for your needs (app name, deployment target, main interface, etc..).




Important thing here is to add Camera permission for this app. In Visual Studio there is no options to add this permission from here. You need to open your **Info.plist** file in some text editor and add this lines:

```
<key>NSCameraUsageDescription</key>
<string>Camera used for scanning</string>
```

Also if you use MX Device as reader device add this protocols in Info.plist:

```
<key>UISupportedExternalAccessoryProtocols</key>
<array>
  <string>com.cognex.dmcc</string>
  <string>com.demo.data</string>
</array>
```

## View Controller

The **ViewController** will be our first controller in **Main.storyboard**. Here we are creating some UI elements and variables that will be used later in this controller.

```
public partial class ViewController : UIViewController, ICMBReaderDeviceDelegate
    {
        protected ViewController(IntPtr handle) : base(handle)
        {
            // Note: this .ctor should not contain any initialization logic.
        }

        CMBReaderDevice readerDevice;
        public bool isScanning = false;

        private NSMutableArray tableData;
        private MXResultsTableSource tableSource;

        public override void ViewDidLoad()
        {
            base.ViewDidLoad();

            ...
```

**lblConnection** - Label UI element for current connection status.

**tableSource** - UITableViewSource for results that will be read.

**btnScan** - Button UI element that will trigger **StartScanning** or **StopScanning**.

**ivPreview** - **ImageView** UI element for showing the last frame of a preview or scanning session.

**CMBReaderDevice** - cmbSDK object that will present MX Device or Phone Camera depends of our configuration.

## Configure ReaderDevice

Here we override the **ViewWillAppear** method to configure reader device object when this view will appear.

If we want to use MX Device for scanning we are using

```
readerDevice = CMBReaderDevice.ReaderOfMXDevice();
```

The availability of the MX Device can change when the device turns ON or OFF, or if the USB cable gets connected or disconnected, and is handled by the ***ICMBReaderDeviceDelegate*** interface. We set this interface as property for reader device with

```
readerDevice.WeakDelegate = this;
```

and allow us to listen these three events:

```
public void DidReceiveReadResultFromReader(CMBReaderDevice reader, CMBReadResults readResults)
public void AvailabilityDidChangeOfReader(CMBReaderDevice reader)
public void ConnectionStateDidChangeOfReader(CMBReaderDevice reader)
```

If we want to configure reader device as Mobile Camera.

```
readerDevice = CMBReaderDevice.ReaderOfDeviceCameraWithCameraMode(CDMCameraMode.NoAimer, CDMPreviewO
ption.Defaults, ivPreview);
```

The **CameraMode** parameter is of the type **CDMCameraMode**, and it accepts one of the following values:

- **NoAimer**: Initializes the reader to use a live-stream preview ( on the mobile device screen ) so the user can position the barcode within the camera's field of view for detection and decoding. Use this mode when the mobile device does not have an aiming accessory.
- **PassiveAimer**: Initializes the reader to use a passive aimer, which is an accessory that is attached to the mobile device or mobile device case that uses the built-in LED flash of the mobile device as a light source for projecting an aiming pattern. In this mode, no live-stream preview is presented on the device screen, since an aiming pattern will be projected.
- **FrontCamera**: Initializes the reader to use the front facing camera of the mobile device, if available (not all mobile devices have a front camera). This is an unusual, but possible configuration. Most front-facing cameras do not have auto focus and

illumination, and provide significantly lower resolution images. This option should be used with care. In this mode illumination is not available.

All of the above modes provide the following default settings for the reader:

- The rear camera is used.
- The zoom feature isa vailable and a button to control it is visible on the live-streamp review (if displayed).
- The simulated hardware trigger is disabled.
- When ***startScanning()*** is called, the decoding process is started. ( Seek ***CDMPreviewOptionPaused*** for more details.)

Based on the selected mode, the following additional options and behaviors are set:

- **NoAimer**
- The live-stream preview is displayed when the ***startScanning()*** method is called.
- Illumination is available and a button to control it is visible on the live-stream preview.
- If commands are sent to the reader for aimer control, they will be ignored.

- **PassiveAimer**
- The live-stream preview will not be displayed when the ***startScanning()*** method is called.
- Illumination is not available and the live-stream preview will not have an illumination button.
- If commands are sent to the reader for illumination control, they will be ignored, since it is assumed in this mode that the built-in LED of the mobile device is being used for the aimer.

- **FrontCamera**
- The live-stream preview is displayed when the ***startScanning()*** method is called.
- The front camera is used.
- Illumination is not available, and the live-stream preview will not have an illumination button. If commands are sent to the reader for aimer or illumination control, they will be ignored.

The **previewOptions** parameter (of type **CDMPreviewOption**) is used to change the reader's default values or override defaults derived from the selected **CameraMode**. Multiple options can be specified by OR-ing them when passing the parameter. The available options are the following:

- **Defaults**: Use this option to accept all defaults set by the **CameraMode**.
- **NoZoomBtn**: This option hides the zoom button on the live-stream preview, preventing a user from adjusting the zoom of the mobile device camera.
- **NoIllumBtn**: This hides the illumination button on the live-stream preview,

preventing a user from toggling the illumination.

- **HwTrigger**: This enables a simulated hardware trigger (the volume down button )for starting scanning on the mobile device. This button only starts scanning when pressed. It does not need to be held like a more traditional purpose-built scanner's trigger. Pressing the button a second time does not stop the scanning process.
- **Paused**: If using a live-stream preview, when this option is set, the preview will be displayed when the ***startScanning()*** method is called, but the reader will not start decoding (i.e. looking for barcodes) until the user presses the on-screen scanning button to actually start the scanning process.
- **AlwaysShow**: This forces alive-stream preview to be displayed, even if an aiming mode has been selected (e.g. **CameraMode == PassiveAimer** )

The last parameter of the type **UIView** is optional and is used as a container for the camera preview. If the parameter is left nil, a full screen preview will be used.

## Connecting to Device

After configuring **ReaderDevice** we need to connect to the device.

```
readerDevice.ConnectWithCompletion((error) => {
   if (error != null)
   {
         new UIAlertView("Failed to connect", error.Description, null, "OK", null).Show();
   }
});
```

If there is some error while trying to connect error will be thrown as parameter in callback function. If everything is fine error parameter will be null.

This function will trigger ***ConnectionStateDidChangeOfReader*** method. If connection is successful **reader.ConnectionState == ConnectionState.Connected.**

After successful connection we can set some settings for ReaderDevice. ReaderDevice settings can be set with already wrapped functions or directly with sending commands to the configured device.

For example if Mobile Camera is used as a ReaderDevice there are **no symbologies enabled by default**. You must enable the symbologies that you want to use with the **SetSymbology** wrapped function.

In this example we are enable some symbologies and set setting to get the last frame from scanning in **ivPreview ImageView**.

```
readerDevice.SetSymbology(CMBSymbology.DataMatrix, true, (error) =>
{
    if (error != null)
```

```
    {
          System.Diagnostics.Debug.WriteLine("FALIED TO ENABLE [DataMatrix], ", error.LocalizedDescr
iption);
    }
});
 readerDevice.SetSymbology(CMBSymbology.Qr, true, (error) =>
 {
    if (error != null)
    {
          System.Diagnostics.Debug.WriteLine("FALIED TO ENABLE [Qr], ", error.LocalizedDescription);
    }
 });
readerDevice.SetSymbology(CMBSymbology.C128, true, (error) =>
{
   if (error != null)
   {
         System.Diagnostics.Debug.WriteLine("FALIED TO ENABLE [C128], ", error.LocalizedDescription)
;
   }
});
readerDevice.SetSymbology(CMBSymbology.UpcEan, true, (error) =>
{
   if (error != null)
   {
         System.Diagnostics.Debug.WriteLine("FALIED TO ENABLE [UpcEan], ", error.LocalizedDescriptio
n);
   }
 });

readerDevice.ImageResultEnabled = true;
readerDevice.SVGResultEnabled = true;
readerDevice.DataManSystem.SendCommand("SET IMAGE.SIZE 0");
```

## Scanning Barcodes

With a properly configured reader, you are now ready to scan barcodes. This can be done by calling the **startScanning** method from your **ReaderDevice** object.

What happens next is based on the type of Reader Device and how it has been configured, but in general:

- If using an MX Device, the user can now press a trigger button on the device to turn the scanner on and read a barcode;
- If using the camera reader, the **cmbSDK** starts the camera, displays the configured live-stream preview, and begins analyzing the frames from the video stream, looking for a configured barcode symbology.

Scanning stops under one of the following conditions:

- The reader found and decoded a barcode;
- The user released the trigger or pressed the stop button on the live-stream preview

screen;
- The camera reader timed out with out finding a barcode;
- The application itself calls the ***stopScanning()*** method.

When a barcode is decoded successfully (the first case), you will receive a **ReadResults** iterable result collection object in **ReaderDevice** listener method.

If your MX Device is configured to work with multi code scanning, you can access all the scanned results from the **results.SubResults** property which is an array that contains **ReaderResult** objects and it will be **null** if single code scanning is used.

## Example

```
public void DidReceiveReadResultFromReader(CMBReaderDevice reader, CMBReadResults readResults)
        {
            btnScan.SetTitle("START SCANNING", UIControlState.Normal);
            isScanning = false;

            tableData.RemoveAllObjects();

            if (readResults.SubReadResults != null && readResults.SubReadResults.Length > 0)
            {
                tableData.AddObjects(readResults.SubReadResults);
                tvResults.ReloadData();

            }
            else if (readResults.ReadResults.Length > 0) {
                tableData.Add(readResults.ReadResults[0]);
            }

            tableSource.SetItems(tableData);
            tableSource.displayResult(0);

            tvResults.ReloadData();
            tvResults.SelectRow(NSIndexPath.FromRowSection(0,0), false, UITableViewScrollPosition.No
ne);
        }
```

## Disconnecting from Device

There may be cases when a device disconnects due to low battery condition or manual cable disconnection. These cases can be detected by the ***ConnectionStateDidChangeOfReader*** callback of the ***ICMBReaderDeviceDelegate***.

**Note:** The ***AvailabilityDidChangeOfReader*** method is also called when the device becomes physically unavailable. It means that the (re)connection is not possible. Always check the availability property of the **ReaderDevice** object before trying to call the ***ConnectWithCompletion*** method.

## Licensing the SDK

If you plan to use the cmbSDK to do mobile scanning with a smartphone or a tablet (with no MX mobile terminal), then the SDK requires the installation of a license key. Without a license key, the SDK will still operate, although scanned results will be obfuscated (the SDK will randomly replace characters in the scan result with an asterisk character).

Contact your Cognex Sales Representative for information on how to obtain a license key including trial licenses which can be used for 30 days to evaluate the SDK.

After obtaining your license key there is two ways to add your license key in application.

The first one is to add it as a key with a value in the project specific **Info.plist** file:

```
<key>MX_MOBILE_LICENSE</key>
<string>Your license key</string>
```

And the second way to implement an activation is directly from the code when you create your readerDevice:

```
....
readerDevice = CMBReaderDevice.ReaderOfDeviceCameraWithCameraMode(CDMCameraMode.NoAimer, CDMPreviewO
ption.Defaults, ivPreview, "YOUR_MX_MOBILE_LICE");
```

## Xamarin.Forms

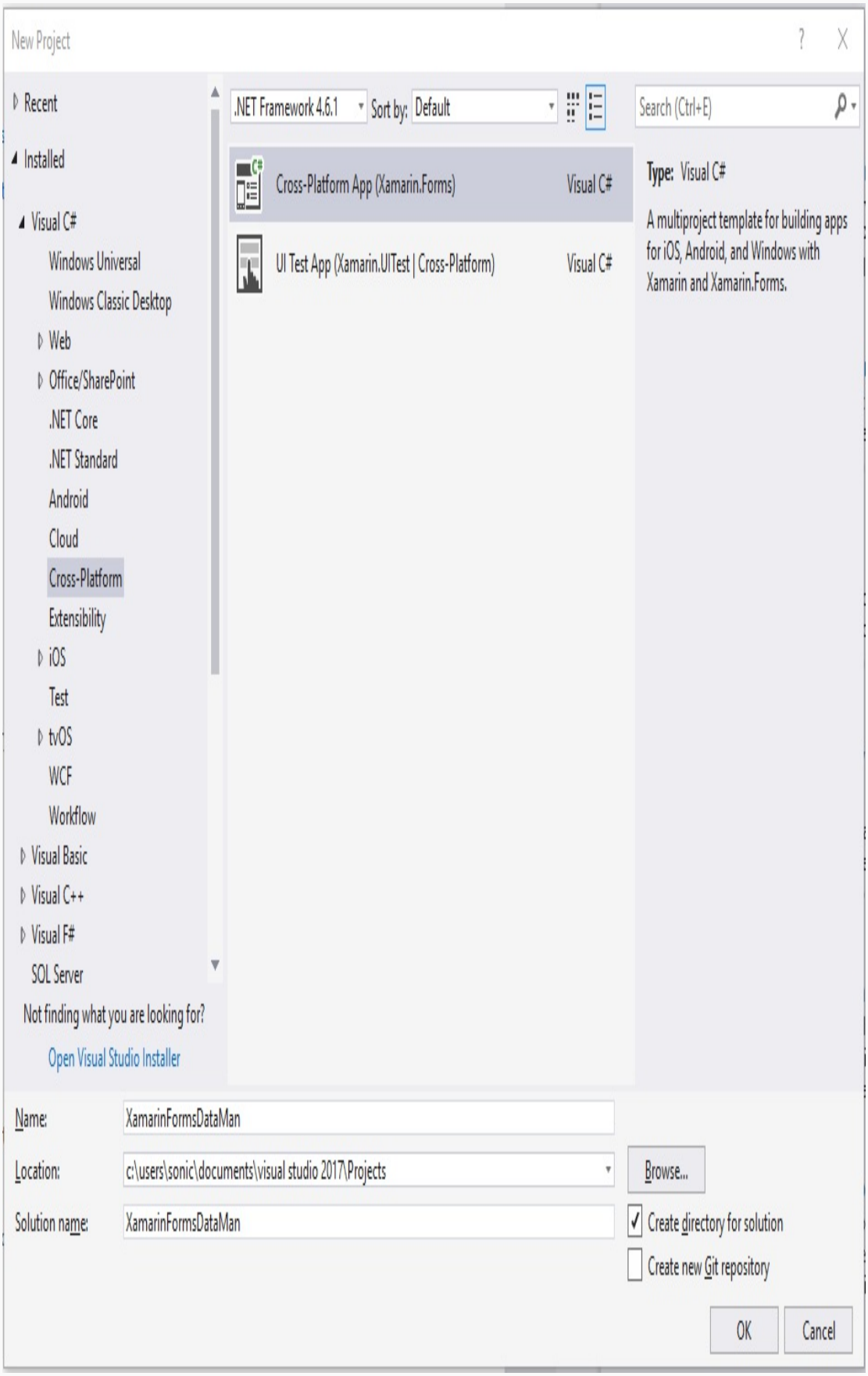
## Getting Started

In the following sections we will explain how our sample app is developed step by step.

Open Visual Studio and follow these steps:

1. Go to **File -> New -> Project.**

2. Create **Cross-Platform App(Xamarin.Forms).**




After loading, see Android Getting Started section to see how to setup references, and the **manifest.xml** for the **Android** platform, or iOS Getting Started and **Info.plist** file

for the **iOS** platform.

## Portable Project

In the portable project we create one page (**MainPage**) where we create a layout for a scanning page and one class(CameraPreview.cs) that will inherit from View control and have some properties and events.

**CameraPreview** control is added in **MainPage**:

```
...

<Grid x:Name="gridCamera">
         <local:CameraPreview x:Name="cameraPreview" SelectedDevice="MobileCamera" ResultReceived="C
ameraPreview_ResultReceived" ConnectionStateChanged="CameraPreview_ConnectionStateChanged" />
         <Label x:Name="lblStatus" Text=" Disconnected " TextColor="White" FontSize="11" VerticalOpt
ions="Start" HorizontalOptions="End" HorizontalTextAlignment="Center" BackgroundColor="#ff4444" Marg
in="0,2,5,0" />
</Grid>

...
```

In **Android** and **iOS** platform specific projects we have custom renderers (ScannerView.cs) for this class, and with that we can use native elements in portable project.

## Custom Renderer

**ViewRenderer** in Android platform specific project:

```
protected override void OnElementChanged(ElementChangedEventArgs<CameraPreview> e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null || Element == null)
            {
                return;
            }

            rlMainContainer = new RelativeLayout(Context);
            rlMainContainer.SetMinimumHeight(50);
            rlMainContainer.SetMinimumWidth(100);
            rlMainContainer.LayoutParameters = new RelativeLayout.LayoutParams(RelativeLayout.Layout
Params.MatchParent, RelativeLayout.LayoutParams.MatchParent);
```

```
            ivPreview = new ImageView(Context);
            ivPreview.SetMinimumHeight(50);
            ivPreview.SetMinimumWidth(100);
            ivPreview.LayoutParameters = new RelativeLayout.LayoutParams(RelativeLayout.LayoutParams
.MatchParent, RelativeLayout.LayoutParams.MatchParent);
            ivPreview.SetScaleType(ImageView.ScaleType.FitCenter);

            rlMainContainer.AddView(ivPreview);

            if (Control == null)
                SetNativeControl(rlMainContainer);

            MainActivity.instance.setActiveReader(Control, Element);
        }
```

In **MainActivity** in Android platform specific project we are handling with reader device object (almost the same like ScannerActivity for **Xamarin.Android**) and from custom renderer we are just call setActiveReader method and pass Control(native RelativeLayout element) and Element(CameraPreview object which is initialized from MainPage).

**ViewRenderer** in iOS platform specific project:

```
protected override void OnElementChanged(ElementChangedEventArgs<XamarinFormsDataMan.CameraPreview>
e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null || Element == null)
            {
                return;
            }

            container = new UIView();
            ivPreview = new UIImageView();
            ivPreview.ContentMode = UIViewContentMode.ScaleToFill;
            ivSVG = new UIImageView();
            ivSVG.ContentMode = UIViewContentMode.ScaleToFill;

            container.AddSubview(ivPreview);
            container.AddSubview(ivSVG);

            ivPreview.Frame = new CoreGraphics.CGRect(0, 0, container.Frame.Size.Width, container.Fr
ame.Size.Height);
            ivPreview.AutoresizingMask = UIViewAutoresizing.FlexibleHeight | UIViewAutoresizing.Flex
ibleWidth;

            ivSVG.Frame = new CoreGraphics.CGRect(0, 0, container.Frame.Size.Width, container.Frame.
Size.Height);
            ivSVG.AutoresizingMask = UIViewAutoresizing.FlexibleHeight | UIViewAutoresizing.Flexible
Width;

            if (Control == null)
                SetNativeControl(container);

            AppDelegate.instance.setActiveReader(Control, Element);
```

```
        }
```

In **AppDelegate** in iOS platform specific project we are handling with reader device object (almost the same like View Controller for **Xamarin.iOS**) and from custom renderer we are just call setActiveReader method and pass Control(native UIImageView element) and Element(CameraPreview object which is initialized from MainPage).

**Configuring ReaderDevice, Connecting to Device, Scanning Barcodes** and **Disconnecting from Device** are the same and you can read about them in Xamarin.Android and Xamarin.iOS sections.

## Licensing the SDK

Licensing the SDK must also be implemented separately in Android and in iOS projects.

For the Android oriented solution please check this link, and for the iOS solution you can refer to this resource.