App Starters (v2.1.x)

Xamarin Forms Shopping Cart

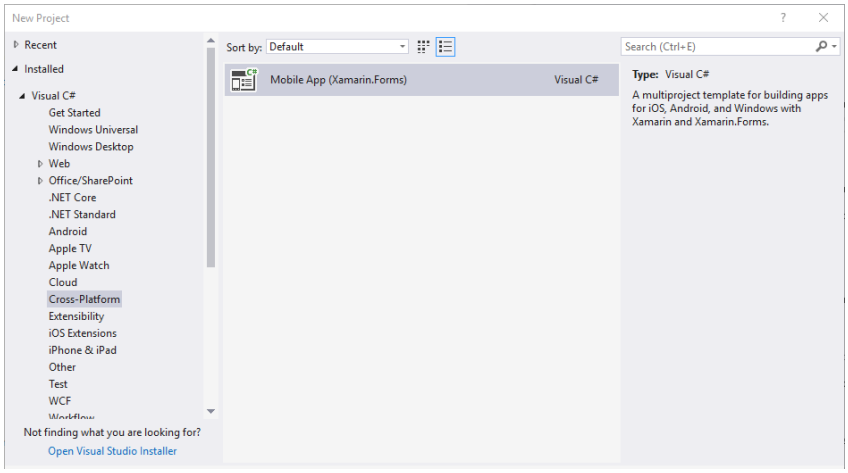
Getting Started

To start developing Xamarin application first you need to install Visual Studio or Xamarin Studio and make sure to include all necessary Xamarin components. For this example we will use Visual Studio. On this link you can read step by step how to download and install Visual Studio for Xamarin applications.

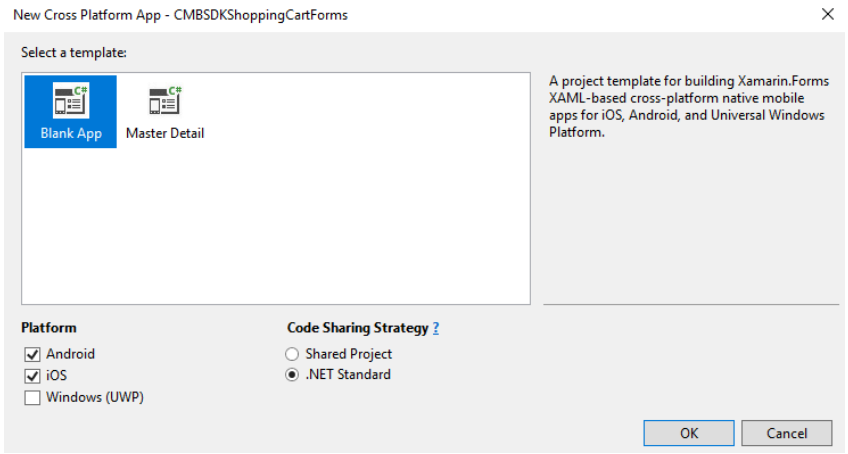
Once you finish with installation open Visual Studio and follow this steps:

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

2. Create **Mobile App (Xamarin.Forms).**



3. Select **Blank App** template for **Android** and **iOS** platform and **.NET Standard** code sharing strategy.



This solution contain three projects:

1.Portable Class Libraries (PCL) Project

A Portable Class Library (PCL) is a special type of project that can be used across disparate CLI platforms such as Xamarin.iOS and Xamarin.Android, as well as WPF, Universal Windows Platform, and Xbox. The library can only utilize a subset of the complete .NET framework, limited by the platforms being targeted.

2. Android Platform-Specific Application Project

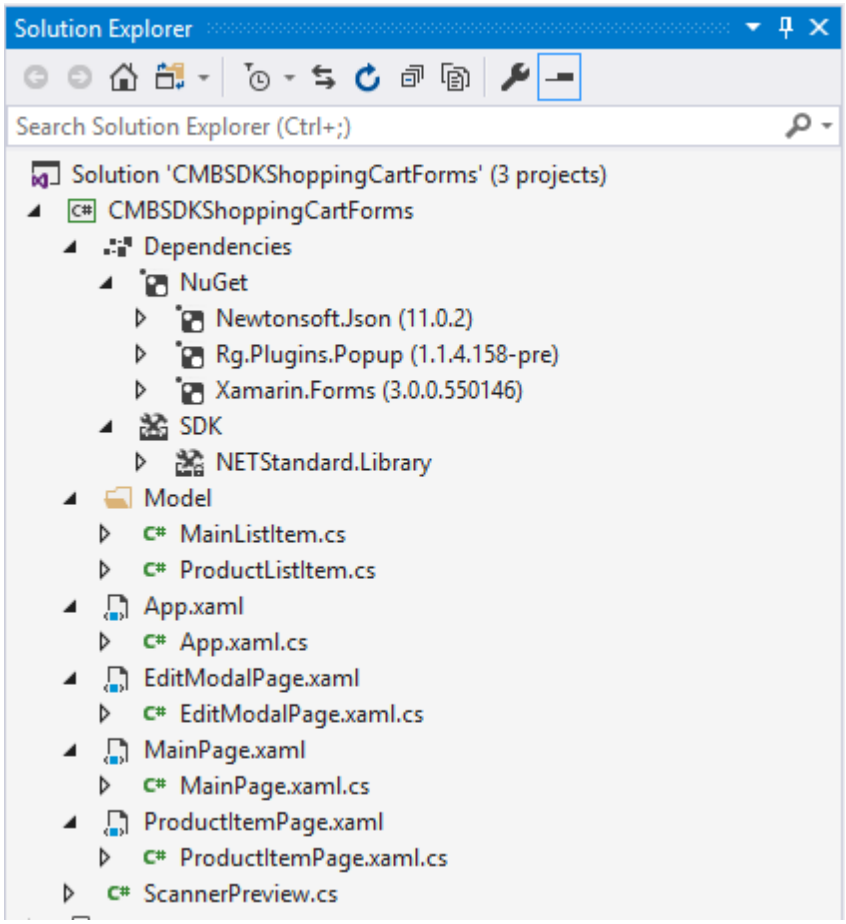
Android platform-specific project must reference the assemblies required to bind Xamarin.Android platform SDK as well as the Core shared code project.

3. iOS Platform-Specific Application Project

iOS platform-specific project must reference the assemblies required to bind Xamarin.iOS platform SDK as well as the Core shared code project.

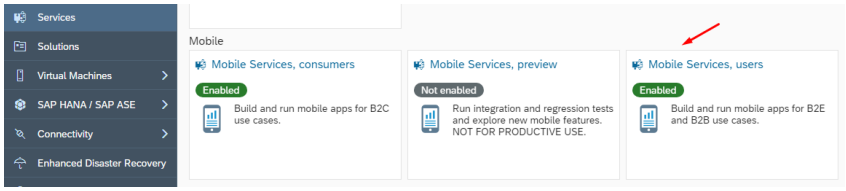
Portable Class Libraries (PCL) Project

In portable project we have **MainPage** where all main lists are presented, **ProductItemPage** where are shown all products for specific list, **EditModalPage** is custom popup to edit list/product name, **MainListItem** and **ProductListItem** models that present list items for lists/products, and **ScannerPreview** custom View control.



ScannerPreview is class that inherit from Xamarin.Forms.View control and we add some additional custom properties and event handlers. Later we will implement custom renderer for this class in platform specific projects.

Here we are using **ScannerPreview** control in **ProductItemPage** to add new or edit existing item.



"gridCamera" is container where we will add **ScannerPreview** control from code behind when this page appear.

For this purpose we will override **OnAppearing()** method on **ProductItemPage**.

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

            if (scannerControl == null)
            {
                scannerControl = new ScannerPreview();

                //You can choose device from here, or you can select on every appearing on this page. Default is SelectFrom
                scannerControl.SelectedDevice = ScanningDevice.MobileCamera;

                //Preview enable. Default is true
                //scannerControl.ScanningPreviewEnable = false;

                //Event that will be triggered when result is received
                scannerControl.ResultReceived += scannerControl_ResultReceived;

                //Event that will be triggered when connection state will be changed
                scannerControl.ConnectionStateChanged += scannerControl_ConnectionStateChanged;

                //Add this control in this content
                gridCamera.Children.Insert(1, scannerControl);

                /*If you use scanner in navigation page and you add another page in navigation stack(where you will use sca
                 * from this page like Navigation.PushAsync(new AnotherPage()) please use this code before you navigate */
                //if (scannerControl != null)
                //{
                //    scannerControl.ResultReceived -= scannerControl_ResultReceived;
                //    scannerControl.ConnectionStateChanged -= scannerControl_ConnectionStateChanged;
                //    gridCamera.Children.Remove(scannerControl);
                //    scannerControl = null;
                //}
                //Navigation.PushAsync(new AnotherPage());
            }
        }
```

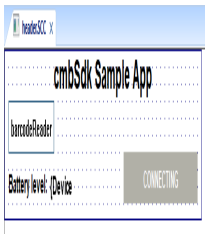
With **scannerControl.ToggleScanning()** we start or stop scanning depends of current state.

With **scannerControl.SelectDevice()** popup window is shown from where we can select which device will be used for scanning (MX Scanner or Mobile Camera).

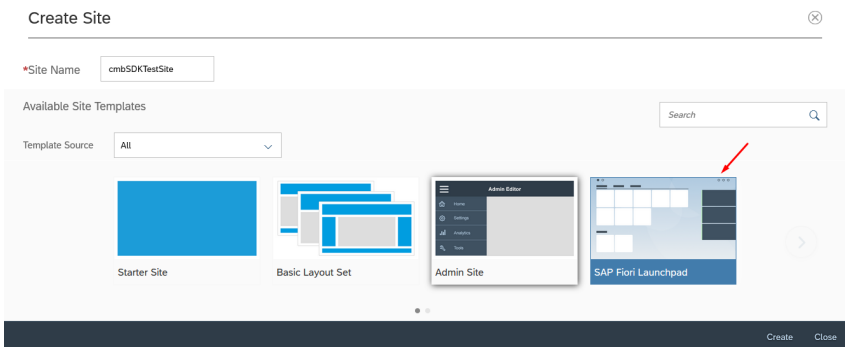
Android Platform-Specific Project

In this project we will set all settings that we need for android platform (min android version, target android version, app name, package name ..), require permissions that we need, add resources for android platform , create custom renderers for android platform , etc..

First we will check **Camera** as required permission for this application.



Next will add resources(that we use in portable project) in drawable folder.



*At the time when this document is written, there is bug for Xamarin Forms with navigation bar icon on android platform, that's why in this project we have small modification on **Toolbar.axml** and there is custom renderer for Navigation Page(**CustomNavigationRenderer**).*

ScannerPreviewRenderer class is custom renderer for **ScannerPreview(PCL custom control)** for Android platform.

```
[assembly: Xamarin.Forms.ExportRenderer(typeof(CMBSDKShoppingCartForms.ScannerPreview), typeof(CMBSDKShoppingCartForms.Droi
namespace CMBSDKShoppingCartForms.Droid
{
    public class ScannerPreviewRenderer : ViewRenderer<CMBSDKShoppingCartForms.ScannerPreview, RelativeLayout>
    {
        private RelativeLayout rlMainContainer;
        private ImageView ivPreview;

        public ScannerPreviewRenderer(Context context) : base(context)
        {
        }

        protected override void OnElementChanged(ElementChangedEventArgs<CMBSDKShoppingCartForms.ScannerPreview> e)
        {
            base.OnElementChanged(e);

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

            //Native container that will be used for live preview while scanning
            rlMainContainer = new RelativeLayout(Context);
            rlMainContainer.SetMinimumHeight(50);
            rlMainContainer.SetMinimumWidth(100);
            rlMainContainer.LayoutParameters = new RelativeLayout.LayoutParams(RelativeLayout.LayoutParams.MatchParent, Rel

            //Native image where will be shown last preview frame
            ivPreview = new ImageView(Context);
            ivPreview.SetMinimumHeight(50);
            ivPreview.SetMinimumWidth(100);
            ivPreview.LayoutParameters = new RelativeLayout.LayoutParams(RelativeLayout.LayoutParams.MatchParent, RelativeL
            ivPreview.SetScaleType(ImageView.ScaleType.FitCenter);

            rlMainContainer.AddView(ivPreview);

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

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

        protected override void Dispose(bool disposing)
        {
            if (Element != null)
                MainActivity.instance.releaseReader(true, Element);

            base.Dispose(disposing);
        }
    }
}
```

Later we will explain **setActiveReader()** and **releaseReader()** methods from **MainActivity** class.

MainActivity

MainActivity is the startup Activity for the application. Here we will configure our **ReaderDevice,** connect to device, listening for availability/connection state change, handling result received, asking for permission, etc..

To listen availability and connection state changing, and to handle request permission result, MainActivity inherit from **IOnConnectionCompletedListener**, **IReaderDeviceListener** and **IOnRequestPermissionsResultCallback**.

setActiveReader() method has two input parameters. First is RelativeLayout which will be used for scanning preview, and second is ScannerPreview element.

Because we can use scanner control in more than one page we have **releaseReader()** method where we release some objects, remove event handlers, and disconnect from reader device, so we can access reader device from another page.

In **setActiveReader()** function after we initialize some variables and set event handlers we call **initDevice()** function.

Here depends of device we select for scanning we call

```
readerDevice = GetMXDevice(this);

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

for **MX Scanner.** The availability of the MX Scanner 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 **Mobile Camera** we can use:

```
readerDevice = GetPhoneCameraDevice(this, CameraMode.NoAimer, PreviewOption.Defaults | PreviewOption.HardwareTrigger, rlMai
```

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).

The last parameter of the type **ViewGroup** is container that we create in renderer for the camera preview.

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);
```

To enable sending the last triggered image from the reader we use:

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

and with

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

we mark and show result from scanned barcode on image.

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 Scanner, 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 Scanner 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)
        {
            //To prevent from out of memory exception
            GC.Collect();

            //List that will be send in Element OnResultReceived that is implemented in portable project
            List<ScannedResult> resList = new List<ScannedResult>();

            //reseting last triggered image
            ivPreview.SetImageBitmap(null);

            //If your MX Scanner is configured to work with multi code scanning, you can access all the scanned results fro
            //property which is an array that contains ReaderResult objects and it will be null if single code scanning is
            if (results.SubResults != null && results.SubResults.Count > 0)
            {
                for (int i = 0; i < results.SubResults.Count; i++)
                {
                    if (results.SubResults[i].IsGoodRead)
                    {
                        Symbology sym = results.SubResults[i].Symbology;
                        if (sym != null)
                        {
                            resList.Add(new ScannedResult(results.SubResults[i].ReadString, sym.Name, true));
                        }
                        else
                        {
                            resList.Add(new ScannedResult(results.SubResults[i].ReadString, "UNKNOWN SYMBOLOGY", true));
                        }
                    }
                    else
                    {
                        resList.Add(new ScannedResult("NO READ", "", false));
                    }

                    if (results.SubResults[i].Image != null)
                    {
                        ivPreview.SetImageBitmap(renderSvg(results.SubResults[i].ImageGraphics, results.SubResults[i].Image
                    }
                    else
                    {
                        if (results.SubResults[i].ImageGraphics != null)
                        {
                            ivPreview.SetImageBitmap(renderSvg(results.SubResults[i].ImageGraphics, ivPreview.Width, ivPrev
                        }
                        else
                            ivPreview.SetImageBitmap(null);
                    }
                }
            }
            else
            if (results.Count > 0)
            {
                ReadResult result = results.GetResultAt(0);

                if (result.IsGoodRead)
                {
                    Symbology sym = result.Symbology;
                    if (sym != null)
                    {
                        resList.Add(new ScannedResult(result.ReadString, sym.Name, true));
                    }
                    else
                    {
                        resList.Add(new ScannedResult(result.ReadString, "UNKNOWN SYMBOLOGY", true));
                    }
                }
                else
                {
                    resList.Add(new ScannedResult("NO READ", "", false));
                }

                if (result.Image != null)
                {
                    ivPreview.SetImageBitmap(renderSvg(result.ImageGraphics, result.Image));
                }
                else
                {
```

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

            isScanning = false;

            //Triggering Element OnResultReceived event that is implemented in portable project
            if (element != null)
                element.OnResultReceived(resList);
        }
```

Disconnecting from Device

Here we override the **OnRestart** and the **OnStop** events so we can do the **Disconnect** and the ***StopAvailabilityListening*** to release all connection when we minimize and to **Initialize** when we restart this activity.

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

            //Check if we are on page where we have ScannerPreview element
            if (element != null)
                initDevice();
        }

        protected override void OnStop()
        {
            //Check if readerDevice is initialized
            if (readerDevice != null)
            {
                readerDevice.StopAvailabilityListening();
                listeningForUSB = false;
                readerDevice.SetReaderDeviceListener(null);
                readerDevice.Disconnect();
                readerDevice = null;
            }

            base.OnStop();
        }
```

iOS Platform-Specific Project

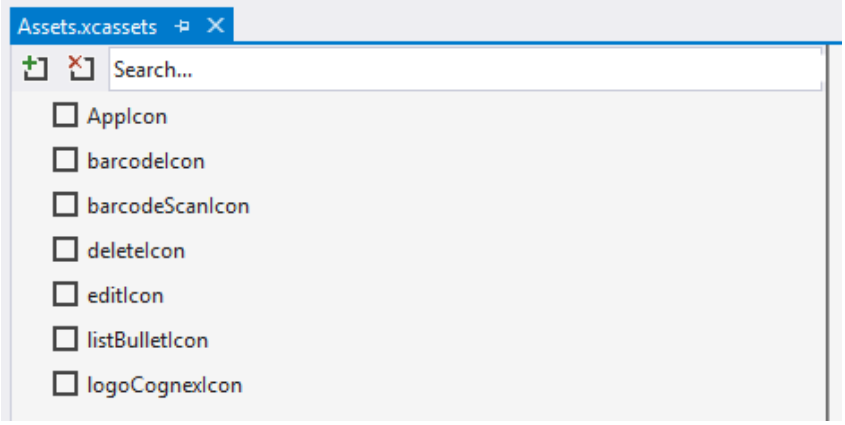
In this project we will set all settings that we need for ios platform (deployment target, app name, bundle identifier ..), require permissions that we need, add assets, create custom renderers for ios platform , etc..

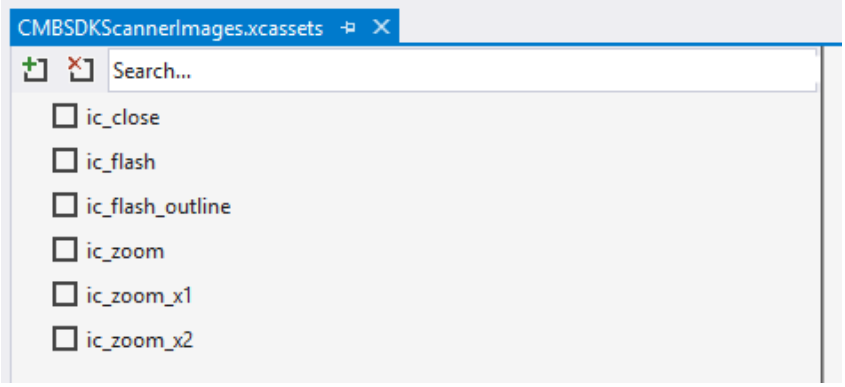
Open **Info.plist** file with some text editor and add following lines:

```
<key>NSCameraUsageDescription</key>
<string>Camera used for scanning</string>
<key>UISupportedExternalAccessoryProtocols</key>
<array>
 <string>com.cognex.dmcc</string>
 <string>com.demo.data</string>
</array>
```

NSCameraUsageDescription key is for camera permission and if we want to use MX Scanner as reader device with this application we should add **UISupportedExternalAccessoryProtocols** key in Info.plist file for this project.

Next will add new icons in Assets catalog and will create one more asset catalog named **CMBSDKScannerImages** and add icons in that catalog





ScannerPreviewRenderer class is custom renderer for **ScannerPreview(PCL custom control)** for iOS platform.

```
[assembly: Xamarin.Forms.ExportRenderer(typeof(CMBSDKShoppingCartForms.ScannerPreview), typeof(CMBSDKShoppingCartForms.iOS.
namespace CMBSDKShoppingCartForms.iOS
{
    public class ScannerPreviewRenderer : ViewRenderer<ScannerPreview, UIView>
    {
        private UIView container;
        private UIImageView ivSVG;
        private UIImageView ivPreview;

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

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

            container = new UIView();
            //Native image that will be used for live preview while scanning
            ivPreview = new UIImageView();
            ivPreview.ContentMode = UIViewContentMode.ScaleToFill;
            //Native image that will be used for SVG image. That will be image on which scaned barcode will bi marked and r
            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.Frame.Size.Height);
            ivPreview.AutoresizingMask = UIViewAutoresizing.FlexibleHeight | UIViewAutoresizing.FlexibleWidth;
```

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

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

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

        protected override void Dispose(bool disposing)
        {
            if (Element != null)
                AppDelegate.instance.releaseReader(true, Element);

            base.Dispose(disposing);
        }
    }
}
```

Later we will explain **setActiveReader()** and **releaseReader()** methods from **AppDelegate** class.

In this class we will configure our **ReaderDevice,** connect to device, listening for availability/connection state change, handling result received, etc..

To listen availability and connection state changing AppDelegate class inherit from **ICMBReaderDeviceDelegate** interface.

setActiveReader() method has two input parameters. First is UIView native control which contains two images, and second is ScannerPreview element.

Because we can use scanner control in more than one page we have **releaseReader()** method where we release some objects, remove event handlers, and disconnect from reader device, so we can access reader device from another page.

In **setActiveReader()** function after we initialize some variables and set event handlers we call **createReaderDevice()** function.

Depends of selected device for scanning in **createReaderDevice()** function we call

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

if we want to use **MX Scanner.** The availability of the MX Scanner 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 AvailabilityDidChangeOfReader(CMBReaderDevice reader)
public void ConnectionStateDidChangeOfReader(CMBReaderDevice reader)
public void DidReceiveReadResultFromReader(CMBReaderDevice reader, CMBReadResults readResults)
```

If we want to configure reader device as **Mobile Camera** we use

```
readerDevice = CMBReaderDevice.ReaderOfDeviceCameraWithCameraMode(CDMCameraMode.NoAimer, CDMPreviewOption.Defaults | CDMPre
```

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 container that we create in renderer for the camera preview.

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 (element != null)
              element.OnConnectionStateChanged("Disconnected");
     }
 });
```

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** image and to mark barcode in **ivSVG image**.

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

            if (element != null && element.ScanningPreviewEnable)
            {
                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 Scanner, 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 **CMBReadResults** iterable result collection object in **ReaderDevice** listener method.

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

Example

```
public void DidReceiveReadResultFromReader(CMBReaderDevice reader, CMBReadResults readResults)
        {
            isScanning = false;

            //List that will be send in Element OnResultReceived that is implemented in portable project
            List<ScannedResult> resList = new List<ScannedResult>();

            //reseting last triggered image
            if (ivPreview != null)
                ivPreview.Image = null;

            if (ivSVG != null)
                ivSVG.Image = null;

            //If your MX Scanner is configured to work with multi code scanning, you can access all the scanned results fro
            //property which is an array that contains ReaderResult objects and it will be null if single code scanning is
            if (readResults.SubReadResults != null && readResults.SubReadResults.Length > 0)
            {
                for (int i = 0; i < readResults.SubReadResults.Length; i++)
                {
                    if (((CMBReadResult)readResults.SubReadResults[i]).XML != null)
                    {
                        XmlDocument xml = new XmlDocument();
                        xml.LoadXml(((CMBReadResult)readResults.SubReadResults[i]).XML.ToString(NSStringEncoding.UTF8));

                        XmlNodeList nll = xml.GetElementsByTagName("status");

                        if (nll.Item(0).InnerText == "GOOD READ")
                        {
                            byte[] base64result = Convert.FromBase64String(xml.GetElementsByTagName("full_string").Item(0).

                            resList.Add(new ScannedResult(System.Text.Encoding.UTF8.GetString(base64result), xml.GetElement
                        }
                        else
                        {
                            resList.Add(new ScannedResult("NO READ", " ", false));
                        }
                    }
                    else
                    {
                        resList.Add(new ScannedResult("NO READ", " ", false));
                    }

                    if (((CMBReadResult)readResults.SubReadResults[i]).Image != null)
                    {
                        if (ivPreview != null)
                            ivPreview.Image = ((CMBReadResult)readResults.SubReadResults[i]).Image;
                    }

                    if (((CMBReadResult)readResults.SubReadResults[i]).ImageGraphics != null)
                    {
```

```
                        if (ivSVG != null)
                        {
                            var parser = new CMBSVG.CDMSVGParser(((CMBReadResult)readResults.SubReadResults[i]).ImageGraphi
                            var svgData = parser.Parse;
                            var renderer = new CMBSVG.CDMSVGRenderer(svgData);
                            UIImage svgImage = renderer.ImageFromSVGWithSize(((CMBReadResult)readResults.SubReadResults[i])

                            ivSVG.Image = svgImage;
                        }
                    }
                }
            }
            else
                foreach (CMBReadResult readResult in readResults.ReadResults)
                {
                    if (readResult.XML != null)
                    {
                        XmlDocument xml = new XmlDocument();
                        xml.LoadXml(readResult.XML.ToString(NSStringEncoding.UTF8));

                        XmlNodeList nll = xml.GetElementsByTagName("status");

                        if (nll.Item(0).InnerText == "GOOD READ")
                        {
                            byte[] base64result = Convert.FromBase64String(xml.GetElementsByTagName("full_string").Item(0).

                            resList.Add(new ScannedResult(System.Text.Encoding.UTF8.GetString(base64result), xml.GetElement
                        }
                        else
                        {
                            resList.Add(new ScannedResult("NO READ", " ", false));
                        }
                    }
                    else
                    {
                        resList.Add(new ScannedResult("NO READ", " ", false));
                    }

                    if (readResult.Image != null)
                    {
                        if (ivPreview != null)
                            ivPreview.Image = readResult.Image;
                    }

                    if (readResult.ImageGraphics != null)
                    {
                        if (ivSVG != null)
                        {
                            var parser = new CMBSVG.CDMSVGParser(readResult.ImageGraphics);
                            var svgData = parser.Parse;
                            var renderer = new CMBSVG.CDMSVGRenderer(svgData);
                            UIImage svgImage = renderer.ImageFromSVGWithSize(readResult.Image != null ? readResult.Image.Si

                            ivSVG.Image = svgImage;
                        }
                    }
                }

            //Triggering Element OnResultReceived event that is implemented in portable project
            if (element != null)
                element.OnResultReceived(resList);
        }
```

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 (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.

For **Android Platform** open your manifest file and add this meta tag inside application tag

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

or you can register your sdk directly from code in your MainActivity before you make connection

```
private void initDevice()
{
   ...

ReaderDevice.SetCameraRegistrationKey("YOUR_MX_MOBILE_LICENSE");
readerDevice.Connect(this);

   ...
}
```

For **iOS Platform** open Info.plist file and add this key

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

or you can register your sdk directly from code in your AppDelegate before you make connection

```
private void createReaderDevice()
{
   ...

   CMBReaderDevice.SetCameraRegistrationKey("Your license key");
   connectToReaderDevice();

   ...
}
```