Nativescript (v2.2.x)

Changelog

Version 1.0.0

- Initial release
- Using cbmSDK v2.2.1 for android and v2.2.3 for iOS

Install cmbSDK Nativescript plugin in your application

From the command prompt go to your app's root folder.

You can use our plugin form npm (recommended) or you can download from here and use from local path.

```
tns plugin add cmbsdk-nativescript # or local path
```

Implement cmbSDK Nativescript plugin in your application

The best way to explore the usage of the plugin is to check our demo app. You can download our demo app from here.

Once you download navigate to demo app root folder and run demo application:

```
cd cmbSDK_Nativescript/demo
tns run android # or ios
```

In short to use our plugin in your project here are the steps:

Import the plugin

- Typescript

```
import { CMBReader, CMBReaderConstants } from 'cmbsdk-nativescript';
```

Set necessary callbacks and configure reader device

Open **home-view-model.ts** from demo app to check this code. All code in our demo app is with short description.

Start scanning process

```
this.cmbReader.startScanning()
```

License Key(s)

IMPORTANT

Usage of the cmbSDK nativescript plugin with an MX device is free, but if you want to utilize the CAMERA DEVICE (scan with the smartphone camera), you need to obtain a license from CMBDN.

he Reader still works without a license, but results are randomly masked with * chars.

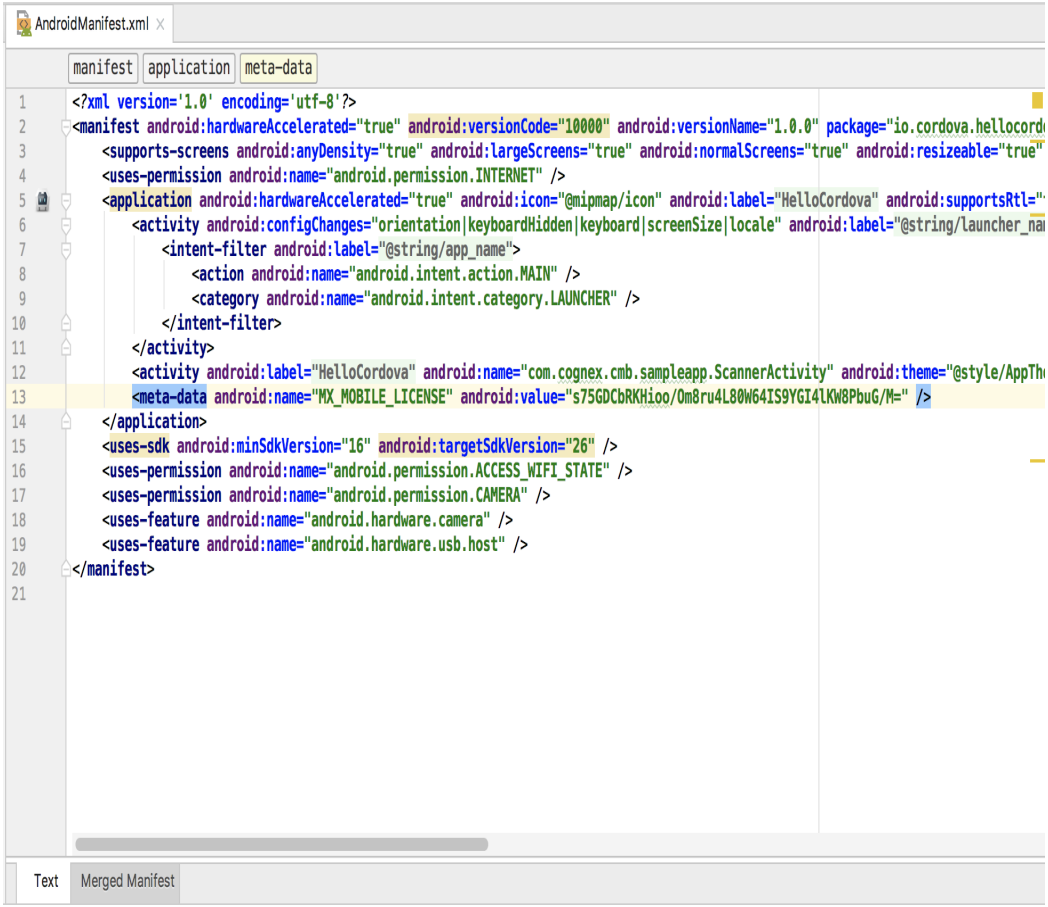
It's free to register and you can obtain a 30 day trial license key.

Once the key is obtained there is two ways to use in your application.

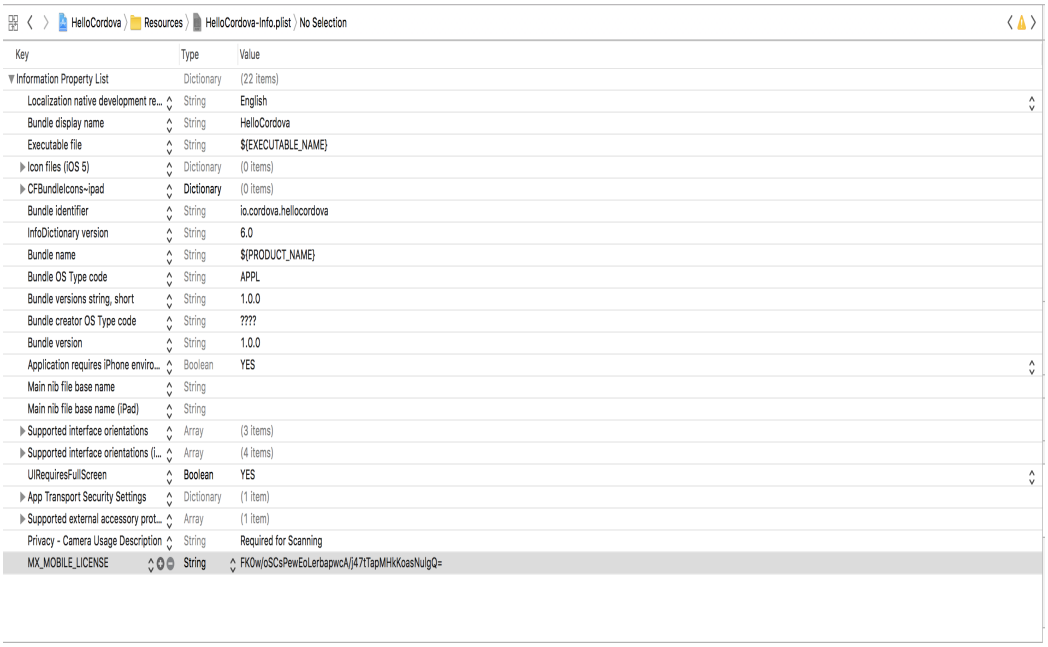
- First way is to include your key from code using **registerSDK** API method. You need to call this method **before** loadScanner.

```
this.cmbReader.registerSDK("SKD_KEY");
```

- Second way is to include your key in AndroidManifest.xml file for Android



and Info.plist file for iOS



API

Methods

(Promise) loadScanner(deviceType: number)

```
/*   @return resolve promise
```

To get a scanner up and running, the first thing to do, is to call the **loadScanner()** method. It expects a device type as input parameter. This method does not connect to the Reader Device. We need to call **connect()** in the promise to actually connect to the Reader Device

```
this.cmbReader.loadScanner(CMBReaderConstants.DEVICE_TYPE.MXReader)
    .then(result => {
        this.cmbReader.connect()
            .then(result => {
                this.updateUIByConnectionState(CMBReaderConstants.CONNECTION_STATE.Connected);
            })
            .catch(err => {
                console.log(err);
                this.updateUIByConnectionState(CMBReaderConstants.CONNECTION_STATE.Disconnected);
            });
        });
    });
```

(Promise) connect()

```
/*   @return
      resolve promise if connection succeded
```

```
      reject promise with error message if there is some problem while connecting
*/
```

The result from the connect() method is returned as a Promise and it will return the result of the connection attempt:

```
this.cmbReader.connect()
    .then(result => {
        this.updateUIByConnectionState(CMBReaderConstants.CONNECTION_STATE.Connected);
    })
    .catch(err => {
        console.log(err);
        this.updateUIByConnectionState(CMBReaderConstants.CONNECTION_STATE.Disconnected);
    });
```

There is Event Listener for the connection status of the ReaderDevice, namely the **CMBReaderConstants.EVENTS.ConnectionStateChanged** event which is explained in more detail below.

(Promise) disconnect()

```
/*   @return
      resolve promise if disconnection succeded
      reject promise with error message if there is some problem while disconnecting
*/
```

Just as there is **connect()**, there is a **disconnect()** method that does the opossite of **connect()** :

```
this.cmbReader.disconnect();
```

Similarly to **connect()**, **disconnect()** too triggers the **CMBReaderConstants.EVENTS.ConnectionStateChanged** event.

(Promise) getConnectionState()

```
/*   @return
      resolve promise with CMBReaderConstants.CONNECTION_STATE value of the reader's current connectionState
      reject promise with error message if there is some problem while we check current connection state
*/
```

(Promise) startScanning() / (Promise) stopScanning()

```
/*   @return
      resolve promise if scanning is started/stopped
      reject promise with error message if there is some problem while start/stop scanning process ex: if readerDevice not
*/
```

startScanning() and stopScanning() methods triggers **CMBReaderConstants.EVENTS.ScanningStateChanged** event that return true if scanner is started and false if is stopped.

After starting the scanner and scanning a barcode, the scan result triggers the **CMBReaderConstants.EVENTS.ReadResultReceived** event.

If you need to get the current connection state, **getConnectionState()** can be used

```
this.cmbReader.getConnectionState()
    .then(connectionState => {
        if (connectionState == CMBReaderConstants.CONNECTION_STATE.Connected) {
          // reader is connected
        }
    })
    .catch(err => {
        console.log(err);
    });
```

(Promise) setSymbologyEnabled(symbology: number, enable: boolean)

```
/*   @return
      resolve promise with result true if symbology is enabled or false if symbology is disabled
      reject promise with error message if there is some problem while we try to enable the symbology
*/
```

Once there is a connection to the Reader, we can enable symbologies by calling **setSymbologyEnabled()**. It expects a int value of the symbol to be enabled and boolean for ON/OFF.

```
this.cmbReader.setSymbologyEnabled(CMBReaderConstants.SYMBOLOGY.DataMatrix, true)
    .then(result => {
        if (result == true)
            console.log("DataMatrix enabled");
        else
            console.log("DataMatrix NOT enabled");
    })
    .catch(err => {
        console.log(err);
    });
```

(Promise) isSymbologyEnabled(symbology: number)

```
/*   @return
      resolve promise with result true if symbology is enabled or false if symbology is disabled
      reject promise with error message if there is some problem while we check symbology status
*/
```

To check if we have a symbology enabled, we use **isSymbologyEnabled()**.

```
this.cmbReader.isSymbologyEnabled(CMBReaderConstants.SYMBOLOGY.DataMatrix)
    .then(result => {
        if (result == true)
            console.log("DataMatrix enabled");
        else
            console.log("DataMatrix NOT enabled");
    })
    .catch(err => {
        console.log(err);
    });
```

(Promise) setLightsOn(on: boolean) / (Promise) isLightsOn()

```
/*   @return
      resolve promise with result true if light is enabled or false if light is disabled
      reject promise with error message if there is some problem while we check light status
*/
```

If we want to enable the flash by default we can use **setLightsOn()** and to check if it is enabled with **isLightsOn()**

(void) setCameraMode(cameraMode: number)

To set how the camera will behave when we use CAMERA device as a barcode Reader we use:

```
this.cmbReader.setCameraMode(CMBReaderConstants.CAMERA_MODE.NoAimer);
/**
Use camera with no aimer. Preview is on, illumination is available.
CMBReader.CAMERA_MODE.NoAimer = 0,

Use camera with a basic aimer (e.g., StingRay). Preview is off, illumination is not available.
CMBReader.CAMERA_MODE.PassiveAimer = 1,

Use camera with an active aimer (e.g., MX-100). Preview is off, illumination is available.
CMBReader.CAMERA_MODE.ActiveAimer = 2,

Use mobile device front camera. Preview is on, illumination is not available.
CMBReader.CAMERA_MODE.FrontCamera = 3
*/
```

*Note: It should be called BEFORE we call **loadScanner()** for it to take effect. Calling it after the scanner was loaded won't do anything if the scanner is loaded.*

(void) setPreviewOptions(previewOptions: number)

Set the overrided preview options.
This function expects an integer that is a result of the ORed result of all the preview options that we want enabled.
Doesn't return a value.
Should be called BEFORE **loadScanner()** (same as cameraMode())\

Example:

```
this.cmbReader.setPreviewOptions(CMBReaderConstants.CAMERA_PREVIEW_OPTION.NoZoomBtn | CMBReaderConstants.CAMERA_PREVIEW_OPT
```

(void) setPreviewOverlayMode(previewOverlayMode: number)

Set the camera overlay mode. You need to do it before loadScanner is called, otherwise it will not work properly ONLY AVAILABLE ON MX-Mobile

Example:

```
this.cmbReader.setPreviewOverlayMode(CMBReaderConstants.OVERLAY_MODE.OM_CMB);
```

(void) setPreviewContainerPositionAndSize(x: number, y: number, w: number, h: number)

```
/*
  @params x, y, w, h
        x,y  : top left position
        w,h  : width and height of the rectangular in precentages of the full container
*/
```

Used only with the phone camera, sets the size and position of the camera preview screen.

Example:

```
this.cmbReader.setPreviewContainerPositionAndSize(0,0,100,50);
//will set the preview to 0,0 and 100% width 50% height
```

(void) setPreviewContainerFullScreen()

Used only with the phone camera, sets the camera preview to start in full screen instead of partial view.

Example:

```
this.cmbReader.setPreviewContainerFullScreen();
//will set the camera preview to start in full screen when startScanning is called
```

(Promise) enableImage(arg: boolean) / (Promise) enableImageGraphics(arg: boolean)

```
/*   @return
      resolve promise with result true if image or imageGrapics is enabled
      reject promise with error message if there is some problem while enabling image result
*/
```

To enable / disable result type returned as image use

```
this.cmbReader.enableImage(true);
```

Same for **enableImageGrapics()**.

```
this.cmbReader.enableImageGraphics(false);
```

(Promise) getDeviceBatteryLevel()

```
/*   @return
      resolve promise with the charge in percentage
      reject promise with error message if there is some problem while checking battery level
*/
```

Helper method to show the battery levels of the connected device. Use it like this:

```
this.cmbReader.getDeviceBatteryLevel()
    .then(result => {
        console.log(result);
    })
    .catch(err => {
        console.log(err);
    });
```

(Promise) resetConfig()

```
/*   @return
      resolve promise if reset is successful
      reject promise with error message if there is some problem while resetting configuration
*/
```

To reset the configuration options we can use **resetConfig**

```
this.cmbReader.resetConfig()
    .then(result => {
    })
    .catch(err => {
        console.log(err);
    });
```

(void) setParser(parserType: number)

Set cmbScanner result parser. This API must be called when readerDevice is loaded and connected

Example:

```
this.cmbReader.setParser(CMBReaderConstants.PARSERS.AUTO);
```

(Promise) sendCommand(commandString: string)

```
/*   @return
      resolve promise with result depends of commandString
      reject promise with error message if there is some problem while executing command
*/
```

Finally, all the methods can be replaced with sending DMCC strings to the READER device. For that we can use our API method **sendCommand**. It can be used to control the Reader completely with command strings.\ More on the command strings can be found here or here

Use it like this:

```
this.cmbReader.sendCommand("SET SYMBOL.POSTNET OFF")
    .then(result => {
        console.log(result);
    })
    .catch(err => {
        console.log(err);
    });
```

Events

Events list

The Nativescript cmbReader object extends Observable object and have events that can be listened and used in application. These events should be set before loadScanner is called.

```
this.cmbReader.on(CMBReaderConstants.EVENTS.ConnectionStateChanged, (args: any) => {

});
```

Here are list of the events:

- CMBReaderConstants.EVENTS.ReadResultReceived
- CMBReaderConstants.EVENTS.AvailabilityChanged
- CMBReaderConstants.EVENTS.ConnectionStateChanged
- CMBReaderConstants.EVENTS.ScanningStateChanged

ReadResultReceived

This event is triggered whenever a scan result is received. The result is a JSON object with this structure:

```
/**
*       results - json array. If you use multicode mode here you will find main result(set of all partial results together
*       subResults - json array of all partial results (if single code mode is uset this array will be empty)
*       xml - string representation of complete result from reader device in xml format

*       results and subResults are json arrays that contains items with this structure:
*   symbology - integer representation of the barcode symbology detected
*   symbologyString - string representation of the barcode symbology detected
*   readString - string representation of barcode
*       goodRead - bool that indicate if barcode is successful scanned
```

```
 *       xml - string representation of partial result in xml format
 *       imageGraphics - string that represent svg image from last detected frame
 *       image - base64 string that contain image from last detected frame
 *       parsedText - string that represent parsed text from the result
 *       parsedJSON - string that represent parsed text in json format from the result
 *       isGS1 - bool that indicate if barcode is GS1 or not
 */
```

AvailabilityChanged

This event is triggered when the availability of the ReaderDevice changes (example: when the MX Mobile Terminal has connected or disconnected the cable, or has turned on or off). The result is an number containing the availability information.

ConnectionStateChanged

This event is triggered when the connection state of the ReaderDevice changes. The result is an number containing the connection information.

ScanningStateChanged

This event is triggered when the scanner state of the ReaderDevice changes. The result is a boolean that is true if the scanning started, or false if it stopped.