Nativescript (v2.5.x)

Changelog

Detailed changelog can be found **here.**

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.cmb.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.

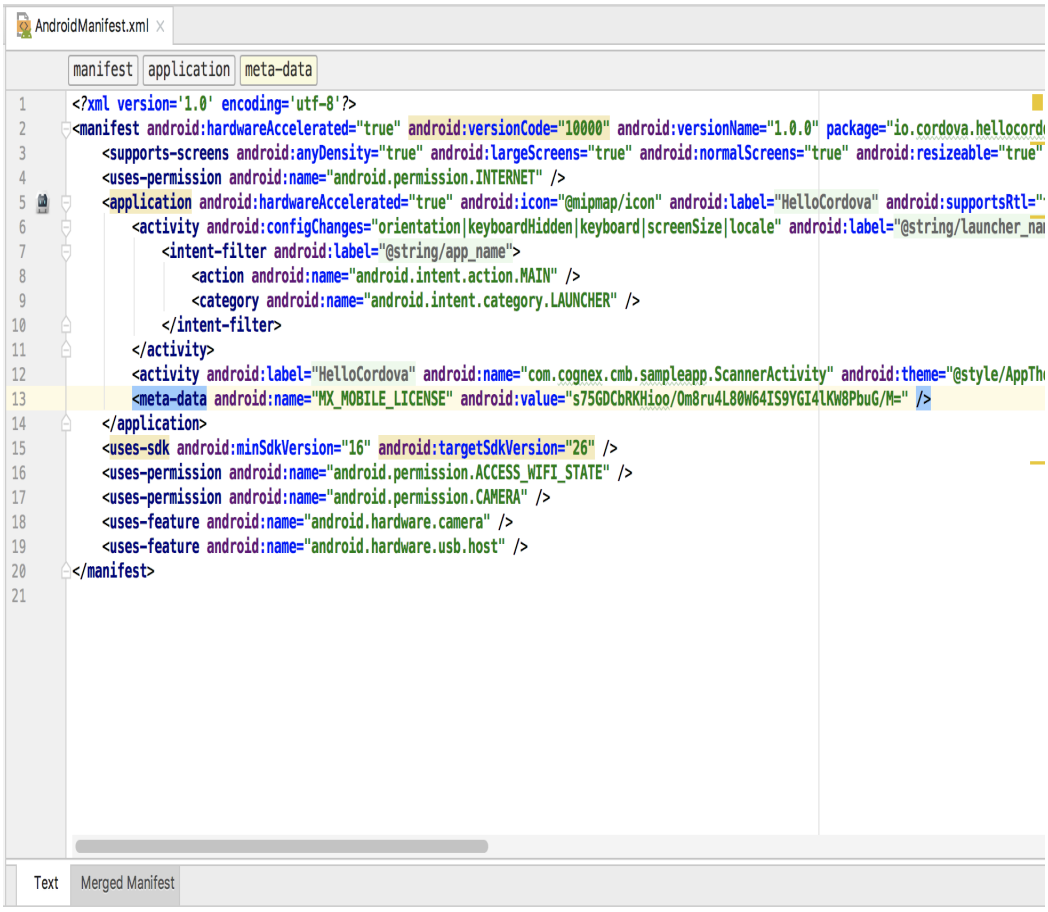
It's free to register and you can obtain a 30 day trial license key.

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

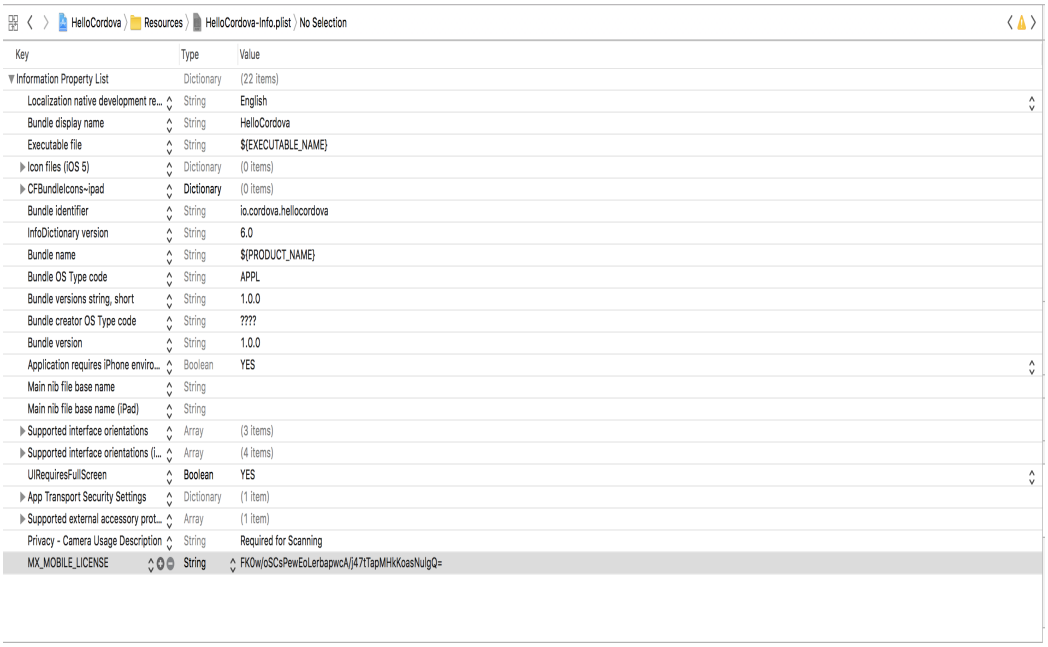
- First way is to include your key from code using **registerSDK** API method. You need to call this method **before** loadScanner.

```
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

loadScanner

To get a scanner up and running, the first thing to do, is to call the **loadScanner()** method. It expects a CMBReaderConstants.DEVICE_TYPE param. This method does not connect to the Reader Device. We need to call **connect()** in the callback to actually connect to the Reader Device

```
this.cmb.loadScanner(CMBReaderConstants.DEVICE_TYPE.MXReader)
    .then(result => {
        this.cmb.connect()
            .then(result => {
                ...
            })
            .catch(err => {
                console.log(err);
            });
    });
```

connect

```
/*   @return
    (promise)  {
        status : boolean, if connection succeeded true if not false
        err : string , if status false err will not be null
    }
*/
```

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

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

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

disconnect

```
/*   @return
    (promise)  {
        status : boolean, if disconnect succeeded true if not false
        err : string , if status false err will not be null
    }
*/
```

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

```
this.cmb.disconnect();
```

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

startScanning

```
/*   @return Promise
      (bool) value of the Scanner Activity (true if the command was successful, false otherwise ex: if readerDevice not ini
*/
```

To start / stop the scanning process, we use these methods. They return a promise, which will be resolved if the command was successful (the scanning has started or stopped) or rejected otherwise (if there is no active ReaderDevice initialized or isn't connected).

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

setSymbologyEnabled

Once there is a connection to the Reader, we can enable symbologies by calling **setSymbologyEnabled()**. It expects three params: a CMBReaderConstants.SYMBOLOGY which is the symbology to be enabled or disabled and a boolean for ON/OFF.

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

isSymbologyEnabled

To check if we have a symbol enabled, we use **isSymbologyEnabled()**. It takes argument CMBReaderConstants.SYMBOLOGY.

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

setLightsOn

```
/*   @return
    (promise)  {
        status : boolean, true if successfully executed command
        err : string , if status false err will not be null
    }
*/
```

If we want to enable the flash we can use **setLightsOn()**. It expects one argument boolean and returns a promise.

isLightsOn

```
/*   @return
    (promise)  {
        status : boolean, true if lights are on, false otherwise
        err : string , in case of error (e.g. reader not initialized)
    }
*/
```

We can check the lights status with isLightsOn(), which returns a promise.

getConnectionState

```
/**
@return A promise that resolves with the CMBReaderConstants.CONNECTION_STATE value of the current reader device
*/
```

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

```
this.cmb.getConnectionState().then(function(connectionState){
  if (connectionState == CMBReaderConstants.CONNECTION_STATE.Connected) {
                // reader is connected
  }
});
```

setPreviewOptions

This should be used only when using the device's built in camera for scanning (CMBReaderConstants.DEVICE_TYPE.Camera).

This function expects one integer argument that is a result of the OR-ed result of all the preview options that we want enabled.

```
this.cmb.setPreviewOptions(CMBReaderConstants.CAMERA_PREVIEW_OPTION.NoZoomBtn | CMBReaderConstants.CAMERA_PREVIEW_OPTION.No
```

*Note: PreviewOptions should be set BEFORE we call **loadScanner()** for it to take effect.*

setPreviewContainerPositionAndSize

This should be used only when using the device's built in camera for scanning (CMBReaderConstants.DEVICE_TYPE.Camera).

setPreviewContainerPositionAndSize takes an array argument with four integer objects, which are the X and Y values for the top left coordinate, and width and height values for the preview container size. All of the values are percentages of the device's screen.

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

setPreviewContainerFullScreen

This should be used only when using the device's built in camera for scanning (CMBReaderConstants.DEVICE_TYPE.Camera).

Sets the camera preview to start in full screen instead of partial view.

```
this.cmb.setPreviewContainerFullScreen();
```

enableImage

Used to enable / disable image result type. Expects one boolean argument.

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

enableImageGraphics

Used to enable / disable svg result type. Expects one boolean argument.

```
this.cmb.enableImageGraphics(true);
```

setParser

Enable or disable parsing for scanned barcodes. Expects one argument of type CMBReaderConstants.RESULT_PARSER.

```
this.cmb.setParser(CMBReaderConstants.RESULT_PARSER.GS1);
```

setReadStringEncoding

Set encoding for the readString result type. Expects one argument of type CMBReaderConstants.READSTRING_ENCODING.

```
this.cmb.setReadStringEncoding(CMBReaderConstants.READSTRING_ENCODING.UTF_8);
```

getDeviceBatteryLevel

```
/*   @return
    (promise)  {
        value : int
        err : string , in case of error (e.g. reader not initialized)
    }
*/
```

Method to show the battery level of the connected device. Doesn't take any arguments.

sendCommand

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. It takes two string arguments, first of which is the DMCC itself, and the second one is to identify it in the response event.

More on the command strings can be found here or here.

```
this.cmb.sendCommand("GET DEVICE.TYPE")
    .then(function (result) {
        console.log(result);
    })
    .catch(function (result) {
        console.log(result);
    });
```

createMDMAuthCredentials

Available only on iOS.

Used for creating authentication credentials used for MDM reporting. It takes four string arguments: username, password, clientID and clientSecret.

Should be called before setMDMReportingEnabled.

More on the MDM Reporting can be found here

```
this.cmb.createMDMAuthCredentials("username", "password", "clientID", "clientSecret");
```

setMDMReportingEnabled

Available only on iOS.

A company owning and operating many Cognex Mobile Terminals may want to remotely collect up-to-date information about battery level, battery health, installed firmware, etc.

An iOS application using the cmbSDK framework can report status information of the attached Mobile Terminal to an MDM instance. This can be enabled with the setMDMReportingEnabled method that accepts one boolean argument.

More on the MDM Reporting can be found here

```
this.cmb.setMDMReportingEnabled(true);
```

Events

The cmbSDK plugin emits Events that can be used in the application.

First, create the event emitter:

```
import NativeModules.NativeEventEmitter;
const scannerListener = new NativeEventEmitter(cmb);
```

and then add listeners for each event you want to handle:

```
this.cmb.on(CMBReaderConstants.EVENTS.AvailabilityChanged, (args: any) => {
    if (args.data == CMBReaderConstants.AVAILABILITY.Available) {
        this.cmb.connect();
    } else {
        dialogs.alert({
            message: "Device became unavailable",
            okButtonText: "OK"
        });
    }
});
```

Here are all the events that the cmbSDK plugin can emit:

```
CMBReaderConstants.EVENT.ReadResultReceived
CMBReaderConstants.EVENT.AvailabilityChanged
CMBReaderConstants.EVENT.ConnectionStateChanged
CMBReaderConstants.EVENT.ScanningStateChanged
```

CMBReaderConstants.EVENT.ReadResultReceived
This event is triggered whenever a scan result is received. A result is a CMBReadResults object (see CMBReadResults class reference).

CMBReaderConstants.EVENT.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 int containing the availability information.

CMBReaderConstants.EVENT.ConnectionStateChanged
This event is triggered when the connection state of the ReaderDevice changes. The result is an int containing the connection information.

CMBReaderConstants.EVENT.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.