BlockID iOS SDK (v1.6.20)

This document is for version 1.6.20

1KOSMOS

Change Log

Date **(sdk_version)**	**Summary**	**Action required** **(for existing app/s)**
Mar 25, 2022 (v1.6.20)	● Introduce capability to register and authenticate FIDO2 security keys (refer pages #85) ● Added error codes for FIDO2 capability (refer page #91)	
Mar 22, 2022 (v1.6.10)	● SDK available as .xcframework	> Refer to the BlockID SDK setup section; > remove **Framework Thinning Run Script** from **Build Phases**
Mar 16, 2022 (v1.6.00)	● Bug fixes related to LiveID scanning ● SDK installation using Cocoapods	Refer to the BlockID SDK setup section (refer page #12)
Feb 15, 2022 (v1.4.98)	● Updated '*SwiftyTesseract*' pod to '*3.1.3*' ● Bug Fixed - LiveID scan should reset when face is out of focus	Change '*SwiftyTesseract*' pod version to '*3.1.3*' (refer page #12)
Feb 4, 2022 (v1.4.97)	● Introduced an optional method to scan LiveID with face liveness check (refer pages #51 and #84)	No action required
Jan 31, 2022 (v1.4.95)	● Introduced feature to bypass PoI (Proof of Identity) based on License module configuration	No action required
Jan 13, 2022 (v1.4.93)	● Introduced feature to verify Social Security Number (SSN) using document verification service (refer pages #50 and #82)	
Dec 20, 2021	● Introduced an ability to set proxy for outbound API calls (page # 19) ● Added Document Verification Service (page #49)	

	○ The SDK now supports Driver License verification API (page #49) ● The LiveIDScannerHelper class now supports an option to NOT to RESET LiveId scan when expression goes wrong (page #51)	
Oct 22, 2021	● Optional parameter ***isDataRequiredOnFail*** added to document (DL, Passport, National ID) scanner helper initializer method (page #23, page #33, page #41) ● Added a public method named ***compareFaces(imgOne: imgTwo: purpose: completion:)*** to allow application to compare two faces (page #61) ● Updated error code section related to invalid document (page #81) ○ removed error code 412 - invalid document ○ new error codes added related to invalid document following format 412XXX	
Oct 13, 2021	● App plist update for RFID scan (page #35) ○ Optional attribute *NFCUserMessageDuringRFIDScan* - add this to app's info.plist if custom message should be displayed during RFID scan	
Oct 11, 2021	● Updated pod version of TrusWalletCore (page #10) ● Start RFID scanning with custom time out (page #35) ● App plist update for RFID scan (page #35)	
Sep 21, 2021	● Below expressions removed in LiveId scan (page #52) ○ MOVE_UP ○ MOVE_DOWN	

	● Below expressions renamed in LiveId scan (page #52) ○ TURN_LEFT (was MOVE_LEFT) ○ TURN_RIGHT (was MOVE_RIGHT)	
Sep 16, 2021	● LiveID Enrollment updated (page #51, #52, #53) ○ Two more Liveness Factors added ■ Move up (move face in up direction) ■ Move down (move face in down direction) ○ Following delegate methods added ■ wrongExpressionDetected ■ liveIdDidDetectErrorInScanning ○ Added Error code 1008 (page #83)	
Aug 24, 2021	● Silent Notification ○ Driver License scanning process will throw a NSNotification named “BlockIDFaceDetectionNotification” which will provide an object with count of faces detected by scanned against “numberOfFaces” (page #32) ○ App can register to this notification, if required any customization on UI ● registerDocument (page #27, #36, #45, #54) ○ New method added to register document which has additional parameter storeArtifact and syncArtifact	
Aug 18, 2021	● LiveID Enrollment updated (page #50 & #51) ○ Following delegate methods are now available,	

	■ focusOnFaceChanged ■ readyForExpression ● Added resetRestorationData() method in restore (page #74) ● Error code changed for kTenantRegisterFailed (page #78) ● New error codes introduced (page #81) ○ public key is not available (kPublicKeyRequired) ○ encryption failed (kEncryption) ○ decryption failed (kDecryption) ● Updated Podfile specification (page #10)	
Aug 6, 2021	● BlockID SDK now supports Enable Bitcode option ● Updated Build Options -> Enable Bitcode -> YES (page #12) ● Added Error code 300 (page #73)	
Jul 26, 2021	● LiveID Enrollment updated Following delegate methods are no longer available, ● focusOnFaceChanged ● readyForExpression Now LiveID works on 4 Liveness Factors ● Blink Eyes ● Smile ● Move Left (move face in left direction) ● Move Right (move face in right direction) (#page 52)	
Jul 20, 2021	● Removed Firebase Analytics pod dependency (page #10)	
Jul 16, 2021	● BlockID SDK bundle name has been updated to **com.onekosmos.blockid.sdk** ● Added new library dependency in pod 'Firebase/Analytics' (page #8)	

	● Changes related to LiveId - mandatory for identity documents and not for miscellaneous documents ● Error message updated for K_LiveId_Wrong_Enrollment (page #79) ● Added Build Options -> Enable Bitcode -> NO (page #12)	
Jul 14, 2021	● sigToken argument in registerDocument() is made Optional now. The value will be nil if not passed to method	
Jul 1, 2021	● sessionId argument in authenticateUser() is an Optional now ● authenticateUser() method callback argument updated, sessionId argument can be Null now, the method returns sessionId ● Stop functions for LiveID, DL and Passport scanning added (were missing from doc only)	
Jun 17, 2021	Document formatting and minor spelling corrections	
Jun 10, 2021	● Added Table of Contents ● Contents formatted and restructured	
Jun 7, 2021	● Classes such as BIDDocumnetData, BIDDriverLiecense, BIDPassport and BIDNationalID are deprecated. Dictionary is being used. The relevant methods below are changed. ● BIDDocumnetType enum has been removed from the register document method changes can be found on the save document section of each document type.	

	● Change in License Checks / Modules ● New section in the SDK functions tile is added to get the server public key. ● Removed typo errors	
Jun 4, 2021	Updated error codes from 429 to 436	
May 27, 2021	cutoutView type changed from UIView to CGRect for document enrollment methods.	
May 21, 2021	● Changes in Document enrollment methods and added new notes before trying to save the data ● Alternate method of saving documents is added ● The Get Document with filter method is added in the document enrollment section. ● Unenrollment document method argument change.	
May 7, 2021	New methods added ● Initialize temp wallet ● Commit Application wallet	
May 6, 2021	OpenSSL universal settings added **NOTE:** BlockID SDK supports only Xcode 12.4 (Device + Simulator)	
Apr 30, 2021	New methods added ● Verify device auth under Device Auth Enrollment ● Verify pin method under Enroll pin Added Openssl, framework information **NOTE**: BlockID SDK supports only device and xcode 12.4	
Mar 10, 2021	RFID	
Mar 10, 2021	Released first draft of this document	

Table of Contents

Overview

This document describes the procedure to configure the BlockID iOS SDK into your application. This integration will allow your users to use the features provided within the iOS SDK of the BlockID mobile application. The features include enrollment of the user’s LiveID (captures the live (real-time) facial images) details, Fingerprint, PIN, Driving License (DL), Passport details and National Id. Also, it has various types of authentication mechanisms by which a user can log in to its system to access it.

Dependencies

You will need the following tools to complete this integration

1. Xcode 13.0 or higher
    a. Deployment target 11.0 and above
    b. Swift 5.5 or higher
2. Cocoapods 1.10.0 or higher
3. SDK is compatible with iOS 11.0 and above

BlockID SDK Setup

The setup consists of the following steps

1. Adding Cocoapods
2. SDK Initialization

Adding CocoaPods

Add the following pods to the podfile of the project and install those pods

```
# Uncomment the next line to define a global platform for your project
 platform :ios, '11.0'

target 'BlockIDTestApp' do
  # Comment the next line if you don't want to use dynamic frameworks
  use_frameworks!

  # Pods for BlockIDTestApp
  pod 'Toast-Swift', '~> 5.0.1'
  pod 'BlockIDSDK', :git =>
'https://dev1kosmos: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@github.com/1KBlockID/ios-b
lockidsdk.git', :tag => '1.6.20'
end

post_install do |installer|
 installer.pods_project.targets.each do |target|
  target.build_configurations.each do |config|

   # set build library for distribution to true
   config.build_settings['BUILD_LIBRARY_FOR_DISTRIBUTION'] = 'YES'

  end
 end
end
```

Run `pod install` command in the terminal. Open .xcworkspace in xcode

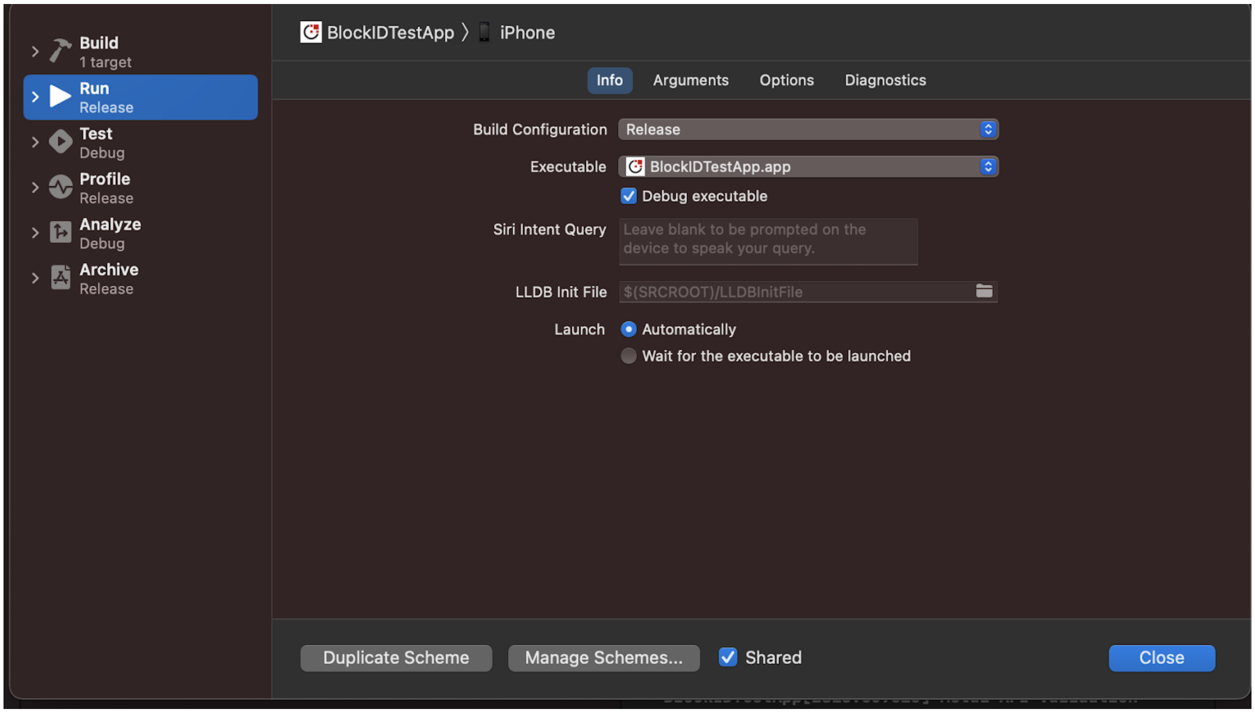
Application Setup

- Go to ‘Build Settings’ of Target
  - Validate Workspace -> Yes
  - Set ‘Enable Bitcode’ option to ‘Yes’

  Note: The above settings must be applied for both Debug and Release Configurations

- Edit scheme from debug to Release (**Optional**).

  Note: The encryption of data takes time while the app runs in debug mode.



Build Configuration: Release

Executable: BlockIDTestApp.app

☑ Debug executable

Siri Intent Query: Leave blank to be prompted on the device to speak your query.

LLDB Init File: $(SRCROOT)/LLDBInitFile

Launch: ● Automatically
○ Wait for the executable to be launched

☑ Shared

SDK Initialization

The SDK is now added to your application, in order to get the features of the SDK, you need to add the license key and tenant details. You have to set following parameters in your project

NOTE: these must be done in the same sequence as listed below

1. Set License key
2. Create wallet
3. Register tenant
4. Commit wallet

NOTE: "set license" function should be called on every launch of the application. Any time the license key is changed (from previous launches), the SDK will self-reset and all stored data will be lost.

Other functions (create, register, commit) should be called **only** if the SDK is not ready. See method `isReady()` below

Set License key

To set the license key use the code below.

```
BlockIDSDK.sharedInstance.setLicenseKey(key: String)
```

Request Parameters

Params	**Description**
key	String value of license key

Initialize wallet

Below function is used to initialize a temporary wallet.

```
 BlockIDSDK.sharedInstance.initiateTempWallet() {(status, error) in
     //your code here
}
```

Response Parameters

Params	Description
status	Boolean value
error	An object of ErrorResponse containing ● code: Integer ● message: String

Register Tenant

Tenants can be represented as the value or an object on which ones api calls can be registered.There are two types of tenant one will be the root and another will be the client tenant. Root tenant will be used to store the details regarding the user enrolled biometric and digital assets where else the users enrolled persona will be stored on the client tenant.

Application is expected to call ‘registerTenant’ once. If an existing tenant is set, the application will load it after validating licenseKey.SDK will only allow tenant registry if and only if the license key is already added.

NOTE: A root tenant can always be a client tenant but a client tenant cannot be a root tenant

Tenant details is the object which will be set in the framework for APIs

```
static let defaultTenant = BIDTenant.makeTenant(tag:
"tenantTag", community: "default", dns: "dns/url")
```

Request Parameters

Params	Description
tag	String value of tenant tag
community	String value of community
dns	String value of DNS

Below function is used to register Tenant details with the framework. These details will be used when APIs will get executed while enrolling documents, or while getting a public key.

When this function gets called, the ethereum wallet gets created and a Distributed Identifier along with 12 Mnemonic phrases (recovery phrases) will be available for the user.

```
 BlockIDSDK.sharedInstance.registerTenant(tenant: BIDTenant)
{(status, error, tenant) in
     //your code here
}
```

Request Parameters

Params	Description
tenant	Type of BIDTenant

Response Parameters

Params	Description
status	Boolean value
error	An object of ErrorResponse containing ● code: Integer ● message: String

tenant	Type of BIDTenant

Commit wallet

Below function is used to commit a temporary wallet. This function will get called on success status of registerTenant

```
BlockIDSDK.sharedInstance.commitApplicationWallet()
```

Get SDK Version

To get the SDK’s current version, the below method should be called. The method returns the current version of the SDK in String format. (e.g. 1.4.4.xxxxxx where xxxxxx is build number generated randomly)

```
BlockIDSDK.sharedInstance.getVersion()
```

Check if the SDK is Ready

If the application license key, wallet and tenant are set, the below method will return true. Application should check this on every launch. If the SDK is not ready, all other functions will fail and may lead to a crash. If the SDK is not ready, the application is responsible for re-initialization (see Initialize Wallet, register tenant, commit wallet).

```
BlockIDSDK.sharedInstance.isReady()
```

Reset SDK

Call the following function if you need to reset the SDK for any reason (eg: “user enters wrong pin 5 times”).

This function purges the SDK wallet, public private key set and any other data stored / owned by the SDK.

```
BlockIDSDK.getInstance().resetSDK()
```

Enable Proxy

BlockID SDK supports adding a proxy to all the outbound API calls. To support this feature, applications can use the below methods.

Set Proxy

```
BlockIDSDK.getInstance().setProxy(host, port, userName, password);
```

Params	Description
host	String value
port	Int value
userName	String value
password	String value

NOTE: This method **must** be called before calling setLicenseKey method

Get proxy details

```
BlockIDSDK.getInstance().getProxyDetails();
```

SDK / Data Security

- To prevent any kind of unauthorized access to the SDK, there is a locking mechanism implemented in the BlockID SDK.
- The SDK will always be in the locked state.To unlock the SDK one has to successfully login/register in any mode. After that, the SDK will be unlocked and all the APIs will be accessible.
- If the SDK is locked and someone tries to access the API, it will show the message as “Unauthorized access”.
- For security purposes, on the app side, the developers have to lock the SDK again, when the app goes in the background or the user is directed to the Login screen.
- Check SDK lock status before calling any enrollment APIs of SDK for security purposes.

The following methods are exposed to the application from SDK

Feature	**Function / Method**
Lock SDK	BIDAuthProvider.shared.lockSDK()
Check if SDK is unlocked	BIDAuthProvider.shared.isSDKUnlocked()
Unlock SDK	BIDAuthProvider.shared.unlockSDK()

SDK Functions

Get Server Public Key

To get the server public key, the below method should be called.The method returns the value of the server public key in String format.

```
BlockIDSDK.sharedInstance.getServerPublicKey()
```

Get Mnemonic Phrases

Mnemonic phrases (recovery phrases) are 12 in count. These phrases are used for restoration of the wallet in future. To get the mnemonic phrases call below function

NOTE: You only need to call this if you need to support specific UX for the user to view / backup mnemonic phrases.

```
BlockIDSDK.sharedInstance.getMnemonicPhrases()
```

Get Distributed Identifier(DID)

Distributed Identifier (DID) is an Identifier which is globally unique. When you create a wallet; it is created along with it. All assets like documents, biometric will be enrolled against this DID.

```
BlockIDSDK.sharedInstance.getDID()
```

Document Enrollment

BlockID SDK allows two main categories of documents: Identity or Misc (Miscellaneous)

NOTE: Before enrolling documents, please ensure that your specific licenseKey has been authorized on the admin console. You can contact the support team for this.

NOTE: All documents must have following baseline attributes

Attribute	Description
id	String (It represents the document's ID eg: driver license number)
category	String (Currently this SDK supports two categories for documents that can be registered i.e, identity_document or misc_document)
type	String (This represents the type of document that the user will try to register e.g.: pin, DL, PPT, liveid, nationalid)
proofedBy	String (This identifies the entity responsible for proofing the document. When you use BlockID SDK Scanners, this defaults to “blockid”)

NOTE: BlockID platform will restrict enrollments to approved proofers. Please contact 1Kosmos support to ensure that your entity names are approved before using another proofer.

Identity Documents baseline attributes

- firstname: String
- lastname: String
- dob: String <yyyyMMdd>
- doe: String <yyyyMMdd>
- face: Base64 string for the face photo
- image: Base64 string for the image of the document (eg: dl front image)

  Note:
  - For documents that carry front and back images, we recommend providing the back image as a Base64 string as well
  - The date params are read from the document and then converted into yyyyMMdd format for further use

Misc. Documents do not have any additional baseline requirements.

BlockID SDK now expects a dictionary to register and unregister documents.

NOTE:

- Before registering an identity document, please make sure that LiveId is enrolled. If LiveId is not already enrolled, the SDK will throw an error “LiveId is mandatory”.
- Please check for camera permissions before using any methods to scan a document.

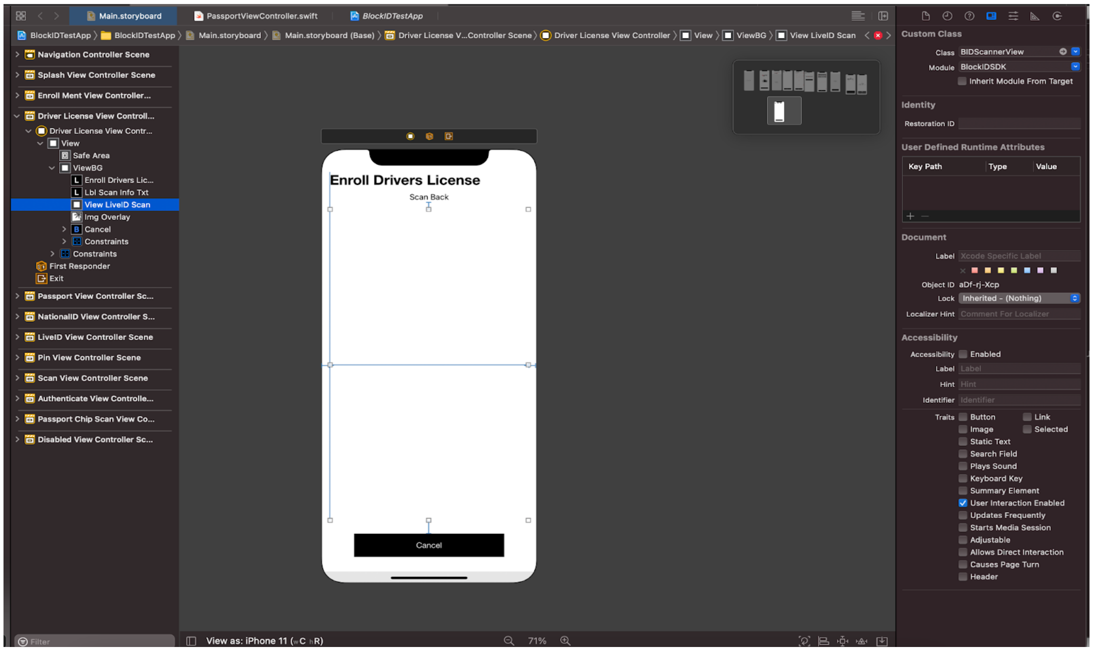
Driver License Enrollment

To scan driver licenses, the app should call the DriverLicenseScanHelper class. This scanner scans the front and backside of the driver license.

There is a parameter for the scanning side of the driverLicense. Based on this parameter the scanner will act and scan the document accordingly.

The front side of the document will be scanned using OCR, and the backside of the document will be scanned for PDF417 barcode.

The BidScannerView is a Scanner view application that can be used to scan documents. The scanner view can be set as a Class in the application. Also you can create a view programmatically.


Main.storyboard
PassportViewController.swift
BlockIDTestApp
BlockIDTestApp 〉 BlockIDTestApp 〉 Main.storyboard 〉 Main.storyboard (Base) 〉 Driver License V...Controller Scene 〉 Driver License View Controller 〉 View 〉 ViewBG 〉 View LiveID Scan
Navigation Controller Scene
Splash View Controller Scene
Enroll Ment View Controller...
Driver License View Controll...
Driver License View Contr...
View
Safe Area
ViewBG
Enroll Drivers Lic...
Lbl Scan Info Txt
View LiveID Scan
Img Overlay
Cancel
Constraints
Constraints
First Responder
Exit
Passport View Controller Sc...
NationalID View Controller S...
LiveID View Controller Scene
Pin View Controller Scene
Scan View Controller Scene
Authenticate View Controlle...
Passport Chip Scan View Co...
Disabled View Controller Sc...
Enroll Drivers License
Scan Back
Cancel
Filter
View as: iPhone 11 (wC hR)
71%

Custom Class

Class: BIDScannerView

Module: BlockIDSDK

□ Inherit Module From Target

Identity

Restoration ID:

User Defined Runtime Attributes

Key Path	Type	Value

Document

Label: Xcode Specific Label

Object ID: aDf-rj-Xcp

Lock: Inherited - (Nothing)

Localizer Hint: Comment For Localizer

Accessibility

Accessibility: □ Enabled

Label: Label

Hint: Hint

Identifier: Identifier

Traits: □ Button □ Link
□ Image □ Selected
□ Static Text
□ Search Field
□ Plays Sound
□ Keyboard Key
□ Summary Element
☑ User Interaction Enabled
□ Updates Frequently
□ Starts Media Session
□ Adjustable
□ Allows Direct Interaction
□ Causes Page Turn
□ Header

OR you can create a UIView programmatically. Use the code below

```
private var _viewLiveIDScan = BIDScannerView()

private func createScannerView() {
     _viewLiveIDScan.frame = CGRect(x: 15, y: 123, width: 384,
height: 645)
       self.view.addSubview(_viewLiveIDScan)
}
```

Method below presents the DriverLicenseScanHelper

```
private var dlScannerHelper: DriverLicenseScanHelper?

private let selectedMode: ScanningMode = .SCAN_LIVE // this will be
.SCAN_LIVE, .SCAN_DEMO

private let firstScanningDocSide: DLScanningSide = .DL_BACK // this
parameter will be .DL_BACK or .DL_FRONT

//initialize the scanner object
```

```
dlScannerHelper = DriverLicenseScanHelper.init(isDataRequiredOnFail:
Bool = false, scanningMode: selectedMode, bidScannerView:
_viewLiveIDScan, dlScanResponseDelegate: self, cutoutView:
_imgOverlay, expiryGracePeriod: 90)
```

Request Parameters

Params	**Description**
scanningMode	Type of ScaningMode The value of the above parameters ● .SCAN_LIVE ● .SCAN_DEMO
bidScannerView	Type of BIDScannerView
dlScanResponseDelegate	viewController
cutoutView	Type of CGRect Frame of the overlay in which the document will be detected.
expiryGracePeriod	Number of days (int) to allow as grace period ahead of document expiry. SDK will throw an error if the document has already expired. IF the document expires before gracePeriod, SDK will complete the scan and return an error <advisory> as well. It is the application's responsibility to decide if to allow enrollment.
isDataRequiredOnFail	An optional Bool parameter with default value - *false* If value is *true* - the app can receive DL data in case of failed scan

Call start driver license scanning

```
self.dlScannerHelper?.startDLScanning(scanningSide:
firstScanningDocSide)
```

Call Stop driver license scanning

```
self.dlScannerHelper?.stopDLScanning()
```

Request Parameters

Params	Description
scanningSide	Type of DLScanningSide: The value of the above parameters .DL_BACK or .DL_FRONT

DriverLicenseResponseDelegates: Extend your controller with this delegate to get a response from the DriverLicense scanner.

```
extension yourViewController: DriverLicenseResponseDelegate {
     //your code here
}
```

Below are delegate methods

- dlScanCompleted
- scanFrontSide
- scanBackSide
- readyForDetection

1. readyForDetection

This method will be called when driver license scan ready for detection

```
func readyForDetection(){
     //your code here
}
```

2. scanBackSide

This method will be called when driver license is ready for backSide Scan

```
func scanBackSide(){
 //your code here
   self.dlScannerHelper?.startDLScanning(scanningSide: .DL_BACK)
}
```

3. scanFrontSide

This method will be called when driver license is ready for frontSide Scan

```
func scanFrontSide(){
   //your code here
   self.dlScannerHelper?.startDLScanning(scanningSide: .DL_FRONT)
}
```

4. dlScanCompleted

This method will be called when driver license scan completed

```
func dlScanCompleted(dlScanSide: DLScanningSide, dictDriveLicense:
[String : Any]?, signatureToken signToken: String?, error:
ErrorResponse?) { //register the document call
BlockIDSDK.sharedInstance.registerDocument, // for implementation see below
}
```

Parameters

Params	Description
dlScanSide	Type of DLScanningSide Values can be ● .DL_BACK

	● .DL_FRONT
dictDriveLicense	Dictionary object<String, Any> with pre-set data of DL including mandatory attributes.
signatureToken	Type of String MD5 value of BIDDriverLicense Object
error	An object of ErrorResponse containing ● code: Integer ● message: String

Register Driver License

To register a driver license object, the BlockIDSDK provides the below method. By default, the method will always store (on device) and sync (to blockchain) the DL document data.

```
BlockIDSDK.sharedInstance.registerDocument(obj: docObject, sigToken:
token){status, error in
        //your code here
}
```

The SDK also provides the below method to register documents which enables application developers to control if the DL document data are required to be stored (on device) and/or synced (to blockchain).

```
BlockIDSDK.sharedInstance.registerDocument(obj:docObject,
storeArtifacts: storeArtifacts, syncArtifacts: syncArtifacts,
sigToken: token){status, error in
        //your code here
}
```

Request Parameters

Params	Description
obj	Dictionary object<String, Any> with pre-set data of DL including mandatory attributes.
signToken	Type of String
storeArtifact	Boolean value to store DL data on device
syncArtifact	Boolean value to sync DL data to EtH / IPFS

Response Parameters

Params	Description
status	Boolean value.
error	An object of ErrorResponse containing ● code: Integer ● message: String

Check if Driver License Enrolled

Below framework method is used to check if one / more Driver Licenses are enrolled / registered with the BlockID.

```
BlockIDSDK.sharedInstance.isDLEnrolled()
```

To get all enrolled Driver License for category identity document use below code

```
let arrDocuments = BIDDocumentProvider.shared.getUserDocument(id:
nil,
type: RegisterDocType.DL.rawValue,
category: RegisterDocCategory.Identity_Document.rawValue)

//Convert json string to array of dictionary
guard let arrDoc =
CommonFunctions.convertJSONStringToJSONObject(strDoc) as? [[String :
Any]] else {
            return
        }

//Get particular document
let dictDoc = arrDoc?.first as? [String: Any]
```

Request Parameters

Params	Description
id	Type String, if you need the whole DL array, send this param null or else you need a particular DL then send the DL's Id.
type	Type of RegisterDocType ● `PPT` ● `DL` ● `NATIONAL_ID` ● `PIN` ● `LIVE_ID`
category	Type of RegisterDocCategory ● `Misc_Document` ● `Identity_Document`

UnEnroll Driver License

To unenroll the driver license, BlockID SDK provides a method for document unregister, details of which are given below.

```
//Call below function to unregister document

BlockIDSDK.sharedInstance.unregisterDocument(dictDoc: <String, Any>)
{
        status, error in
}
```

Request Parameters

Params	Description
dictDoc	Dictionary object <String, Any> with pre-set data of DL including mandatory attributes.

Response Parameters

Params	Description
status	Boolean value.
error	An object of ErrorResponse containing ● code: Integer ● message: String

BIDDocumentProvider class provides the below method if an application tends to retrieve an already enrolled Driver License object.

```
//Get JSON string of document
let strDoc = BIDDocumentProvider.shared.getUserDocument(id: id,
type: registerDocType.rawValue, category: nil) ?? ""

//Convert json string to array of dictionary
guard let arrDoc =
CommonFunctions.convertJSONStringToJSONObject(strDoc) as? [[String :
Any]] else {
            return
        }


//Get particular document

let dictDoc = arrDoc?.first as? [String: Any]
```

Request Parameters

Params	Description
id	Type String, if you need the whole DL array, send this param null or else you need a particular DL then send the DL Id.
type	Type of RegisterDocType ● `PPT` ● `DL` ● `NATIONAL_ID` ● `PIN` ● `LIVE_ID`
category	Type of RegisterDocCategory ● `Misc_Document` ● `Identity_Document`

Number of Faces Captured

BlockIDSDK will throw a NSNotification that can be captured by application or by the entity using BlockIDSDK. This notification will have following configurations

Notification Name	BlockIDFaceDetectionNotification
Notification Object	● “numberOfFaces” - Int `(count of faces)` ● “documentType” - RegisterDocumentType ○ `DL`

Passport Enrollment

To scan a passport, the app should call the PassportScanHelper class. This scanner scans the Passport text as well as MRZ. The below method presents the PassportScanHelper.

```
private var ppScannerHelper: PassportScanHelper?
private let selectedMode: ScanningMode = .SCAN_LIVE

//initialize the scanner object
self.ppScannerHelper = PassportScanHelper.init(isDataRequiredOnFail:
Bool = false, scanningMode: self.selectedMode, bidScannerView:
self._viewLiveIDScan, ppResponseDelegate: self, cutoutView:
self._imgOverlay, expiryGracePeriod: 90)
```

Request Parameters

Params	Description
scanningMode	Type of ScaningMode The value of the above parameters ● .SCAN_LIVE ● .SCAN_DEMO
bidScannerView	Type of BIDScannerView
ppResponseDelegate	viewController
cutoutView	Type of CGRect Frame of the overlay in which the document will be detected.
expiryGracePeriod	Number of days (int) to allow for a grace period ahead of document expiry. **Note**: The SDK will throw an error if the document has already expired. IF the document expires before gracePeriod, the SDK will complete

	the scan and return an error <advisory> as well. It is the application's responsibility to decide if to allow enrollment.
isDataRequiredOnFail	An optional Bool parameter with default value - *false* If value is *true* - the app can receive Passport data in case of failed scan

Call to start Passport scanning

```
self.ppScannerHelper?.startPassportScanning()
```

Call to stop Passport scanning

```
self.ppScannerHelper?.stopPassportScanning()
```

Call to start e-Chip(RFID) scanning of Passport

```
self.ppScannerHelper?.startRFIDScanning(defaultTimeout: TimeInterval = 15)
```

Note: For Passport RFID scan, the app must have below entry in the .plist file

Plist Format
Mandatory ● ISO7816 application identifiers for NFC Tag Reader Session *Key*: Item 0 *Value*: A0000002471001 *Type*: String **Optional** ● NFCUserMessageDuringRFIDScan *Key*: NFCUserMessageDuringRFIDScan *Value*: Hold your iPhone near an NFC enabled passport. *Type*: String
XML Format
Mandatory

```
<key>com.apple.developer.nfc.readersession.iso7816.select-identifiers</key>
<array>
        <string>A0000002471001</string>
</array>
```

Optional

```
<key>NFCUserMessageDuringRFIDScan</key>
        <string>Hold your iPhone near an NFC enabled passport.</string>
```

PassportResponseDelegate

Extend your controller with this delegate to get a response from the Passport scanner

```
extension yourViewController: PassportResponseDelegate {
     //your code here
}
```

Below are delegate methods

- passportScanCompleted
- readyForDetection

1. readyForDetection

This method will be called when passport scan ready for detection

```
func readyForDetection(){
     //your code here
}
```

2. passportScanCompleted

This method will be called when the passport scan is completed.

```
func passportScanCompleted(withBidPassport obj: [String : Any]?,
 error: ErrorResponse?, signatureToken signToken: String?,
isWithRFID: Bool?) {
BlockIDSDK.sharedInstance.registerDocument}
```

Parameters

Params	Description
obj	Dictionary object<String, Any> with pre-set data of PPT including mandatory attributes.
signatureToken	String (MD5 value of BidPassport Object)
error	An object of ErrorResponse containing ● code: Integer ● message: String
isWithRFID	Boolean (true/false)

Register Passport

To register a passport object, the BlockIDSDK provides the below method. By default, the method will always store (on device) and sync (to blockchain) the Passport document data.

```
BlockIDSDK.sharedInstance.registerDocument(obj: docObject, sigToken:
token){status, error in
        //your code here
}
```

The SDK also provides the below method to register documents which enables application developers to control if the Passport document data are required to be stored (on device) and/or synced (to blockchain).

```
BlockIDSDK.sharedInstance.registerDocument(obj:docObject,
storeArtifacts: storeArtifacts, syncArtifacts: syncArtifacts,
sigToken: token){status, error in
        //your code here
}
```

Request Parameters

Params	Description
obj	Dictionary object<String, Any> with pre-set data of Passport including mandatory attributes.
signToken	Type of String
storeArtifact	Boolean value to store Passport data in local storage (if it's true, it will save the data else not)
syncArtifact	Boolean value to sync Passport data with Eth/IPFS (if it's true, it will save the data else not)

Response Parameters

Params	Description
status	Boolean value.
error	An object of ErrorResponse containing ● code: Integer ● message: String

Check if Passport enrolled

Below framework method is used to check if one or more passports are enrolled / registered with the BlockID

```
BlockIDSDK.sharedInstance.isPassportEnrolled()
```

To get all Passport enrolled for

```
let arrDocuments = BIDDocumentProvider.shared.getUserDocument(id:
nil,
type: RegisterDocType.PPT.rawValue,
category: RegisterDocCategory.Identity_Document.rawValue)

 //Convert json string to array of dictionary
guard let arrDoc =
CommonFunctions.convertJSONStringToJSONObject(strDoc) as? [[String :
Any]] else {
            return
        }

//Get particular document
let dictDoc = arrDoc?.first as? [String: Any]
```

Request Parameters

Params	Description
id	Type String, if you need the whole Passport array, send this param null or else you need a particular Passport then send the Passport Id.
type	Type of RegisterDocType ● `PPT` ● `DL` ● `NATIONAL_ID` ● `PIN` ● `LIVE_ID`
category	Type of RegisterDocCategory ● `Misc_Document` ● `Identity_Document`

Unenroll Passport

To unenroll the passport, BlockID SDK provides a method for document unregister, details of which are given below.

```
//Call below function to unregister document

BlockIDSDK.sharedInstance.unregisterDocument(dictDoc: dictDoc) {

        status, error in

}
```

Request Parameters

Params	**Description**
dictDoc	Dictionary object<String, Any> with pre-set data of Passport including mandatory attributes.

Response Parameters

Params	**Description**
status	Boolean value.
error	An object of ErrorResponse containing ● code: Integer ● message: String

BIDDocumentProvider class provides the below method if an application tends to retrieve an already enrolled Passport object

```
//Get JSON string of document
let strDoc = BIDDocumentProvider.shared.getUserDocument(id: id,
type: registerDocType.rawValue, category: nil) ?? ""

//Convert json string to array of dictionary
```

```
guard let arrDoc = 
CommonFunctions.convertJSONStringToJSONObject(strDoc) as? [[String : 
Any]] else {
            return
        }

//Get particular document

let dictDoc = arrDoc?.first as? [String: Any]
```

Request Parameters

Params	Description
id	Type String, if you need the whole Passport array, send this param null or else you need a particular Passport then send the Passport Id.
type	RegisterDocType (`PPT/ DL/ NATIONAL_ID/ PIN/ LIVE_ID)`
category	RegisterDocCategory (`Misc_Document/ Identity_Document)`

NationalID Enrollment

To scan NationalId, the app should call the NationalIDScanHelper class. This scanner scans the front and backside of the document. There is a parameter for the scanning side of the nationalId. Based on this parameter the scanner will act and scan the document accordingly. The front side of the document will be scanned using OCR, and the backside of the document will be used to scan OCR, MRZ, and QRCode. The method below presents the NationalIdScanHelper.

```
private var nationalIDScanHelper: NationalIDScanHelper?

private let selectedMode: ScanningMode = .SCAN_LIVE // this will be
.SCAN_LIVE, .SCAN_DEMO

private let firstScanningDocSide: NIDScanningSide = .NATIONAL_ID_BACK
 // this parameter will be .DL_BACK or .DL_FRONT

//initialize the scanner object
nationalIDScanHelper = NationalIDScanHelper.init(isDataRequiredOnFail:
Bool = false, scanningMode: selectedMode, bidScannerView:
_viewLiveIDScan, nidScanResponseDelegate
: self, cutoutView:_imgOverlay, expiryGracePeriod: 90)
```

Request Parameters

Params	**Description**
scanningMode	Type of ScaningMode The value of the above parameters ● .SCAN_LIVE ● .SCAN_DEMO
bidScannerView	Type of BIDScannerView
nidScanResponseDelegate	viewController
cutoutView	Type of CGRect

	Frame of the overlay in which the document will be detected.
expiryGracePeriod	Number of days (int) to allow for a grace period ahead of document expiry. The SDK will throw an error if the document has already expired. IF the document expires before gracePeriod, the SDK will complete the scan and return an error <advisory> as well. It is the application's responsibility to decide if to allow enrollment.
isDataRequiredOnFail	An optional Bool parameter with default value - *false* If value is *true* - the app can receive NationalID data in case of failed scan

Call function for scanning NationalID

```
self.nationalIDScanHelper?.startNationalIDScanning(scanningSide:
firstScanningDocSide)
```

Request Parameters

Params	**Description**
scanningSide	Type of DLScanningSide The value of the above parameters ● .DL_BACK ● .DL_FRONT

NationalIDResponseDelegates

Extend your controller with this delegate to get a response from the NationalID scanner.

```
extension yourViewController: NationalIDResponseDelegate {
     //your code here
}
```

Below are delegate methods

- nidScanCompleted
- scanFrontSide
- scanBackSide
- readyForDetection

1. readyForDetection

This method will be called when NationalID scan ready for detection

```
func readyForDetection(){
     //your code here
}
```

2. scanBackSide

This method will be called when driver license is ready for backSide Scan

```
func scanBackSide(){
 //your code here
   self.nationalIDScannerHelper?.startNationalIDScanning(scanningSide:
.NATIONAL_ID_BACK)
}
```

3. scanFrontSide

This method will be called when driver license is ready for frontSide Scan

```
func scanFrontSide(){
   //your code here
   self.nationalIDScannerHelper?.startNationalIDScanning(scanningSide:
.NATIONAL_ID_FRONT)
}
```

4. nidScanCompleted

This method will be called when nationalID scan completed

```
func nidScanCompleted(nidScanSide: NIDScanningSide, dictNationalID:
[String : Any]?, signatureToken signToken: String?, error:
ErrorResponse?) { //register the document call
BlockIDSDK.sharedInstance.registerDocument, for implementation see below
}
```

Parameters

Params	Description
nidScanSide	Type of DLScanningSide Values can be ● .NATIONAL_ID_BACK ● .NATIONAL_ID_FRONT
dictNationalID	Dictionary object<String, Any> with pre-set data of National Id including mandatory attributes.
signatureToken	Type of String MD5 value of BIDNationalID Object
error	An object of ErrorResponse containing ● code: Integer ● message: String

Save NationalID

To register a NationalID object, the BlockIDSDK provides the below method. By default, the method will always store (on device) and sync (to blockchain) the NationalID document data.

```
BlockIDSDK.sharedInstance.registerDocument(obj: docObject, sigToken:
token){status, error in
        //your code here
}
```

The SDK also provides the below method to register documents which enables application developers to control if the NationalID document data are required to be stored (on device) and/or synced (to blockchain).

```
BlockIDSDK.sharedInstance.registerDocument(obj:docObject,
storeArtifacts: storeArtifacts, syncArtifacts: syncArtifacts,
sigToken: token){status, error in
        //your code here
}
```

Request Parameters

Params	Description
obj	Dictionary object<String, Any> with pre-set data of NationalID including mandatory attributes.
signToken	Type of String
storeArtifact	Boolean value to store NationalID data in local storage (if it's true, it will save the data else not)
syncArtifact	Boolean value to sync NationalID data with Eth/IPFS (if it's true, it will save the data else not)

Response Parameters

Params	Description
status	Boolean value.
error	An object of ErrorResponse containing ● code: Integer ● message: String

Check if NationalID Enrolled

Below framework method is used to check if one or more NationalID is enrolled / registered with the BlockID.

```
BlockIDSDK.sharedInstance.isNationalIDEnrolled()
```

To get all Passport enrolled for

```
let arrDocuments = BIDDocumentProvider.shared.getUserDocument(id:
nil,
type: RegisterDocType.NATIONAL_ID.rawValue,
category: RegisterDocCategory.Identity_Document.rawValue)

 //Convert json string to array of dictionary
guard let arrDoc =
CommonFunctions.convertJSONStringToJSONObject(strDoc) as? [[String :
Any]] else {
            return
        }

//Get particular document
let dictDoc = arrDoc?.first as? [String: Any]
```

Request Parameters

Params	Description
id	Type String, if you need the whole National Id array, send this param null or else you need a particular National Id then send the National Id's Id.
type	Type of `RegisterDocType` ● `PPT` ● `DL` ● `NATIONAL_ID` ● `PIN` ● `LIVE_ID`
category	Type of `RegisterDocCategory`

	● Misc_Document ● Identity_Document

UnEnroll NationalID

To unenroll the National ID, BlockID SDK provides a method for document unregister, details of which are given below

```
//Call below function to unregister document
BlockIDSDK.sharedInstance.unregisterDocument(dictDoc: dictDoc) {
        status, error in
}
```

Request Parameters

Params	Description
dictDoc	Dictionary object<String, Any> with pre-set data of National Id including mandatory attributes.

Response Parameters

Params	Description
status	Boolean value.
error	An object of ErrorResponse containing ● code: Integer ● message: String

BIDDocumentProvider class provides the below method if an application tends to retrieve an already enrolled National ID object

```
//Get JSON string of document
let strDoc = BIDDocumentProvider.shared.getUserDocument(id: id,
type: registerDocType.rawValue, category: nil) ?? ""

//Convert json string to array of dictionary
```

```
guard let arrDoc =
CommonFunctions.convertJSONStringToJSONObject(strDoc) as? [[String :
Any]] else {
            return
        }

//Get particular document

let dictDoc = arrDoc?.first as? [String: Any]
```

Request Parameters

Params	Description
id	Type String, if you need the whole National Id array, send this parameter null or else you need a particular National Id then send the National Id.
type	Type of RegisterDocType ● `PPT` ● `DL` ● `NATIONAL_ID` ● `PIN` ● `LIVE_ID`
category	Type of RegisterDocCategory ● `Misc_Document` ● `Identity_Document`

Document Verification Service

BlockID SDK provides the functionality to verify data that the user has presented through our partners.

The current version of BlockID SDK supports the below document verification services.

1. DL Verification
2. SSN Verification

The data for the document verification can be obtained using the following means:

1. using the document scan feature of the BlockID SDK for DL Verification
2. through manual entry from the user for SSN Verification

Once the data is obtained, the following steps can be performed to verify the document.

```
// Call below function to verify document
BlockIDSDK.sharedInstance.verifyDocument(dvcID: String, dic: [String:Any], completion:
@escaping ((Bool, [String:Any]?, ErrorResponse?) -> Void)){ (status, dataDic, error) in
  //your code
}
```

Request Parameters

Params	Description
dvcID	String (Document Verification Connector ID)
dic	Dictionary object <String, Any>

Response Parameters

Params	Description
status	Boolean value.
dataDic	Dictionary

error	An object of ErrorResponse containing ● code: Integer ● message: String

Biometric Enrollment

LiveID Enrollment

The application must use the below initialization method to initiate the LiveIDScannerHelper class.

```
public init(scanningMode: ScanningMode, bidScannerView:
BIDScannerView, shouldResetOnWrongExpresssion: Bool = true,
liveIdResponseDelegate: LiveIDResponseDelegate)
```

Request Parameters

Params	Description
scanningMode	ScanningMode ● SCAN_LIVE ● SCAN_DEMO
bidScannerView	Custom UI object of class BIDScannerView
shouldResetOnWrongExpresssion	**true / false - default value is true**
liveIdResponseDelegate	LiveIDResponseDelegate object

BlockID SDK provides the below method to start LiveID scanning.

To start LiveID scanning, use below method

```
LiveIDScannerHelper.startLiveIDScanning(dvcID: String? = nil);
```

Request Parameters

Params	Description
dvcID (Optional)	String (document verification connector id) - If the value passed is not nil, then the faceLiveness check will get started.

To stop LiveID scanning, use below method

```
LiveIDScannerHelper.stopLiveIDScanning();
```

```
//create a liveID scanner object
private var liveIdScannerHelper: LiveIDScannerHelper?

//create a scanning mode object
private let selectedMode: ScanningMode = .SCAN_LIVE

//call function to start liveID scanning
private func startLiveIDScanning() {
//1. Check for Camera Permission
        AVCaptureDevice.requestAccess(for: AVMediaType.video) {
response in
            if !response {
                DispatchQueue.main.async {
                    //2. Show Alert
                }
            } else {
                DispatchQueue.main.async {
                    //3. Initialize LiveIDScannerHelper
                    if self.liveIdScannerHelper == nil {
                        self.liveIdScannerHelper =
LiveIDScannerHelper.init(scanningMode: self.selectedMode,
bidScannerView: self._viewLiveIDScan, liveIdResponseDelegate: self)
                    }
                    //4. Start Scanning
               self.liveIdScannerHelper?.startLiveIDScanning()
                }
            }
        }
}


//Stop LiveID Scanning at any point of time
self.liveIdScannerHelper?.stopLiveIDScanning()
```

LiveIDResponseDelegates: Extend your controller with this delegate to get a response from the liveID scanner.

```
extension yourViewController: LiveIDResponseDelegate {
     //your code here
}
```

- liveIdDetectionCompleted
- focusOnFaceChanged
- readyForExpression
- wrongExpressionDetected
- liveIdDidDetectErrorInScanning

1. liveIdDetectionCompleted

On successful liveID detection, this delegate method will be called. This method has two parameters as described below.

```
func liveIdDetectionCompleted(_ liveIdImage: UIImage?,
signatureToken: String?, error: ErrorResponse?) {
          // your code goes here
}
```

Request Parameters

Params	Description
liveIdImage	Type of UIImage. It returns photos captured during liveness
signatureToken	Type of String It returns the md5 token of the image.
error	An object of ErrorResponse containing ● code: Integer ● message: String

2. focusOnFaceChanged

This will return false when the face moves out of focus.

```
func focusOnFaceChanged(isFocused: Bool?) {
          //your code here
}
```

Request Parameters

Params	Description
isFocused	Bool type. It returns false when the face moves out of focus

3. readyForExpression

This will return a value of type LivenessFactorType whenever the face comes first time in focus and on subsequent evaluation of different facial emotions.

```
func readyForExpression(_ livenessFactor: LivenessFactorType) {
          //your code here
}
```

Request Parameters

Params	Description
livenessFactor	LivenessFactorType type. Values can be ● BLINK ● SMILE ● TURN_LEFT ● TURN_RIGHT ● NONE

4. wrongExpressionDetected

This will return a value of type LivenessFactorType that was performed by user while it wasn’t expected.

```
func wrongExpressionDetected(_ livenessFactor: LivenessFactorType) {
          //your code here
}
```

Request Parameters

Params	Description
livenessFactor	LivenessFactorType type. Values can be ● BLINK ● SMILE ● TURN_LEFT ● TURN_RIGHT ● NONE

5. liveIdDidDetectErrorInScanning

This will return an error detected while scanning the LiveID.

```
func liveIdDidDetectErrorInScanning(error: ErrorResponse?){
          //your code here
}
```

Request Parameters

Params	Description
error	An object of ErrorResponse containing ● code: Integer ● message: String

To save enrolled LiveID into the BlockID environment, the below method registers the LiveID with BlockID through the SDK.

```
BlockIDSDK.sharedInstance.setLiveID(liveIdImage: UIImage,
liveIdProofedBy: String, sigToken: token) {
 (status, error) in
          // your code goes here
}
```

Params	Description
liveIdImage	UIImage captured for LiveID
liveIdProofedBy	String (This identifies the entity responsible for proofing the document. When you use BlockID SDK Scanners, this defaults to “blockid”)
signToken	Type of String

Response Parameters

Params	Description
status	Boolean value.
error	An object of ErrorResponse containing ● code: Integer ● message: String ● Object: Dictionary

Register LiveID with Document

The below method registers the LiveID with BlockID through the SDK. Below function is used when we are enrolling any document but live id is not enrolled then this will automatically enroll live id after particular document enrollment

```
BlockIDSDK.sharedInstance.registerDocument(obj: docObject,
liveIdProofedBy: String, docSignToken: docSignToken, faceImage:
UIImage, liveIDSignToken: token) { [self] (status, error) in
}
```

The SDK also provides the below method to register documents which enables application developers to control if the Passport document data are required to be stored (on device) and/or synced (to blockchain).

```
BlockIDSDK.sharedInstance.registerDocument(obj: docObject,
storeArtifacts: storeArtifacts, syncArtifacts: syncArtifacts
, liveIdProofedBy: String, docSignToken: docSignToken, faceImage:
UIImage, liveIDSignToken: token) { [self] (status, error) in
}
```

Request Parameters

Params	**Description**
obj	Dictionary object<String, Any> with pre-set data of Live Id including mandatory attributes.
liveIdProofedBy	String (This identifies the entity responsible for proofing the document. When you use BlockID SDK Scanners, this defaults to “blockid”)
liveIDSignToken	Type of String
docSignToken	Type of String
faceImage	UIImage captured for LiveID

storeArtifacts	Boolean value to store Document data in local storage (if it’s true, it will save the data else not)
syncArtifacts	Boolean value to store Document data in local storage (if it’s true, it will save the data else not)

Response Parameters

Params	**Description**
status	Boolean value.
error	An object of ErrorResponse containing ● code: Integer ● message: String

Check if the LiveID is enrolled

Below framework method is used to check if LiveID is enrolled / registered with the BlockID

```
BlockIDSDK.sharedInstance.isLiveIDRegisterd()
```

Verify LiveID

Below method is used to verify face with already enrolled face with BlockID

```
BlockIDSDK.sharedInstance.verifyLiveID(image: photo, sigToken:
token) { (status, error) in
          // your code goes here
}
```

Request Parameters

Params	Description
image	Type of UIImage. Send uiimage from the value received from the delegate method 'liveIdDetectionCompleted'
signToken	Type of String Send signatureToken from the value received from the delegate method 'liveIdDetectionCompleted'

Response Parameters

Params	Description
status	Boolean value.
error	An object of ErrorResponse containing ● code: Integer ● message: String

Device Auth Enrollment

To Enroll Device Auth

```
BIDAuthProvider.shared.enrollDeviceAuth {  (success, error, message)
in
           // your code goes here
}
```

Check if the Device Auth is enrolled

```
BlockIDSDK.sharedInstance.isDeviceAuthRegisterd()
```

Response Parameters

Params	Description
success	Boolean value.
error	An object of ErrorResponse containing ● code: Integer ● message: String
message	String value for message

To Unenroll Device auth

```
BlockIDSDK.sharedInstance.unenrollDeviceAuth (success, error,
message) in
       // your code goes here
}
```

Response Parameters

Params	Description
success	Boolean value.
error	NSObject of ErrorResponse The above object consists two parameters ● Code type of integer ● Message type of string
message	String value for message

To Verify Device auth

```
BidAuthProvider.shared.verifyDeviceAuth (success, error, message) in
      // your code goes here
}
```

Response Parameters

Params	Description
success	Boolean value.
error	An object of ErrorResponse containing ● code: Integer ● message: String
message	String value for message

Pin Enrollment

To Enroll Pin

```
BlockIDSDK.sharedInstance.setPin(pin: appPin, proofedBy:
YOUR_PROOFEDBY) {
 (status, error) in
    // your code goes here
}
```

Request Parameters

Params	Description
pin	A string value of application pin
proofedBy	A string value of application proofed by

Response Parameters

Params	Description
success	Boolean value.
error	An object of ErrorResponse containing ● code: Integer ● message: String

Check if the pin is enrolled

```
BlockIDSDK.sharedInstance.isPinRegistered()
```

To Verify Pin

```
BlockIDSDK.sharedInstance.verifyPin(pin: pin) {}
```

Request Parameters

Params	Description
pin	String value of application pin

Response Parameters

Return value
Boolean value.

To Unenroll pin

```
BlockIDSDK.sharedInstance.removePin(pin: pin) { (status, error) in
    // your code goes here
}
```

Request Parameters

Params	Description
pin	String value of application pin

Response Parameters

Params	Description
success	Boolean value.
error	An object of ErrorResponse containing ● code: Integer ● message: String

Compare faces

A public method is now introduced with the name ‘compareFaces()’ which will allow applications to compare two faces.

```
BlockIDSDK.sharedInstance.compareFaces(base64Img1: String,
base64Img2: String, purpose: CompareFacePurpose)
{(bidFaceCompareScore, errorResponse) in
      //your code here
}
```

Request Parameters

Params	Description
base64Img1	base64 string value for image 1
base64Img2	base64 string value for image 2
purpose	An enum of type CompareFacePurpose with 2 options ● *authentication* ● *doc_enrollment*

Response Parameters

Params	Description
bidFaceCompareScore	An object of BIDFaceCompareScore containing ● *minScore*: Double - a minimum value set in client’s tenant as per acceptance criteria ● *confidence*: Double - a value returned by cognitive face comparison API ● *match*: Bool - a value returned by tenant based on *minScore* and *confidence* values ○ *true* when if *confidence >= minScore* ○ *false* when if *confidence < minScore* ● *isIdentical*: Bool - a value returned by cognitive API

errorResponse	An object of ErrorResponse containing ● code: Integer ● message: String

License Checks / Modules

License checks are necessary for getting the state of modules, and whether they are enabled for the license. The methods below return the Modules enabled for the specific enrollments.

To get Biometric Enrollments

```
let enrollments =
BlockIDSDK.sharedInstance.getBiometricAssetEnrollments()
```

To Check Specific Enrollment one can use it in these way

```
//This will return if biometric enrollment is enabled for license
enrollments?.BiometricEnabled

//This will return if liveID enrollment is enabled for license
enrollments?.LiveIDEnabled

//This will return if pin enrollment is enabled for license
enrollments?.PinEnabled
```

To get Digital Assets Enrollment

```
BlockIDSDK.sharedInstance.getDigitalAssetEnrollments()
```

To Check Specific Enrollment one can use it in these way

```
//This will return if dl enrollment is enabled for license
enrollments?.DLEnabled

// This will return if passport enrollment is enabled for license
enrollments?.PPEnabled

//This will return if national id enrollment is enabled for license
enrollments?.NIDEnabled

enrollments?.IdentityEnabled
enrollments?.MiscEnabled
```

User Linking

Magic link (get info, redeem)

When user details are added to the BlockID Admin Console, the console will send an email with a magic link to users. This link helps users to register themselves into the application.

Magic link is a base64 encoded string that carries information such as API, tag, community, code. Decode the magic link using the following JSON object:

```
// MARK:-
class MagicLinkModel: NSObject, Codable {
    var api: String? = ""
    var tag: String? = ""
    var community: String? = ""
    var code: String? = ""

    func getBidOrigin() -> BIDOrigin? {
        let bidOrigin = BIDOrigin()
        bidOrigin.api = self.api
        bidOrigin.tag = self.tag
        bidOrigin.community = self.community
        return bidOrigin
    }
}
```

```
let decodedString = String(data: decodedData, encoding: .utf8)!
Let magicLinkData = CommonFunctions.jsonStringToObject(json:
decodedString) as MagicLinkModel?
```

Call the below function to check if the code is valid or not.

```
BlockIDSDK.sharedInstance.validateAccessCode(code:
magicLinkData.code, origin: magicLinkData.getOrigin()) { (status,
error, response) in
  //your code here
}
```

Request Parameters

Params	Description
code	Type of String
origin	Type Of BidOrigin

Response Parameters

Params	Description
success	Boolean value.
error	An object of ErrorResponse containing ● code: Integer ● message: String

response	Type of
	`AccessCodeValidationResponseDecryptedData`
	var id: String!
	var uuid: String!
	var type: String!
	var tenantId: String!
	var communityId: String!
	var accesscodepayload: AccessCodeResponseDataPayload!
	var status: String!
	public class AccessCodeResponseDataPayload: Codable {
	var userid: String!
	var authType: String!
	var invite_email: String!
	var otp_email: String!
	}

In response, you have to check authType, so that the user onboarding can be decided.

If the **authType is none or OTP**, the application will call the public key of the tenant. The URL for the public key will be formed using the URL provided in the magic link

```
let baseUrl = magicLink?.baseUrl + magicLink?.path
let url = baseUrl + "/publickeys"
let headers = ["Content-Type": "application/json"]
BIDNetworkManager.sharedInstance.makeRequest(requestMethod: .get,
serviceUrl: url, requestBody: nil, requestHeaders: headers) {
(response: NetworkResponseCallback<ServerPublicKeyResponse>) in
    //your code here
}
```

On the success of the above method append base64 encoded Payload to the URL and present the webview with a newly formatted URL.

Once the webview is presented, it will get an OTP on an email/SMS, and that OTP has to be entered into the application. Once the OTP is successfully verified, it will return the scephash and signature token to the application. The below method will be called to onboard the user.

```
BlockIDSDK.sharedInstance.addPreLinkedUser(userId: userid, scep_hash:
scephash, scep_privatekey: signaturetoken, origin: origin) { (status,
error) in
  //your code here
}
```

Response Parameters

Params	**Description**
success	Boolean value.
error	An object of ErrorResponse containing

	● code: Integer ● message: String

If the **authType is authN**, the application should present the screen where the user will provide a password, and that user will get linked against the userid and magiccode. To onboard use the below method

```
BlockIDSDK.sharedInstance.redeemAccessCode(code: magicLinkDataCode,
userId: userid, password: password, origin: bidOrigin, deviceToken:
apnsDeviceToken, lat: locationLatitude, lon: locationLongitude) {
(status, error) in
    //your code here
}
```

Authentication (without SCEP)

There are three types in which users can be linked.

- Account link through web URL.
- Adding a SCEP account.
- Adding a Non-SCEP account.

By Account link through web URL

After scanning the QR code, the QR code payload will consist of the model called BIDOrigin. BIDOrigin will then have a method of **getAuthPage();** by calling this method one will get the auth page. Authpage cannot be null if it's null then needs to show an appropriate error for it. If the auth page contains a URL the app should open the webview once the webview is open the app should show a dialog box of 6 digit alphanumeric code if the code matches with the opened webview URL the user should click `yes it's a match` button on the dialog box and should enter the userid and password of the account which needs to be linked. After authenticating the webview the webview will request the encrypted payload after decrypting the payload one will receive the userid then the user id should be linked by calling the following method.

```
BlockIDSDK.sharedInstance.linkUser(userId: userId, origin: origin,
deviceToken: apnstoken, lon: locationLongitude, lat:locationLatitude)
{ (status, error) in
    // your code here
}
```

Request Parameters

Params	Description
userID	String value of userID received from qr scanning.
bidOrigin	BIDOrigin Object constructed from the data received from the qr scanning.
deviceToken	deviceToken

Response Parameters

Params	Description
status	Boolean value.
error	An object of ErrorResponse containing ● code: Integer ● message: String

By Adding a scep account

If the auth page after scanning contains the native auth schema like `"blockid://authenticate"` and the method is scep (which means userid should have a valid signature token and hash to be able to login in the windows.) When the method is equal to scep then one should ask the user for its AD credentials (username and password) after getting the username and password from the user one should call the following api to validate the user and link the user id.

```
BlockIDSDK.sharedInstance.addNativeAccount(method: method, userId: userid,
password: password, origin: bidOrigin!, deviceToken: devicetoken, lon:
locationLongitude, lat:locationLatitude) { (status, error) in
    // your code here
}
```

Request Parameters

Params	Description
method	String value received from authPage URL
userID	String value of userID received from QR scanning.
password	String value of Password
bidOrigin	BIDOrigin Object constructed from the data received from the qr scanning.
deviceToken	deviceToken

Response Parameters

Params	Description
status	Boolean value.
error	An object of ErrorResponse containing ● code: Integer ● message: String

By adding a Non-scep account

The whole process will be the same as above the only difference will be the method value will be null or something else then it will be considered as a non scep user which will be linked but signature token and hash for windows login will not be present.

Authentication (UWL workflows)

Push Notification

For making this feature available, you should first have to link a user with DID and the user should be logged in once with windows in online mode.

At time of link user or add scep account, the push token must be sent to the API(s)

- Once the device has been registered, the user should be able to unlock windows via Push notification
- On push, If the app is already not open (i.e. shutdown or in background), the user MUST be required to login first
- If app was already running OR if the user has logged in (see above line item), then the application will process the push notification (continue reading)
- Push messages contain origin info
- The application MUST verify that the requested userId for unlock matches the userId and origin of a linked account
- if above is FALSE, show error toast: “Notification is for an account that is not linked to this device”, else continue reading
- Present the consent screen similar to login to windows screen.

QR Scanner

To scan QR for login or for adding an account, you must have access to the QRScannerHelper. This class presents with the QRScanner, it returns the QRScanned data. To use the method,see the below code.

```
//create a qr scanner object
private var qrScannerHelper: QRScannerHelper?

//create a scanning mode object
private let selectedMode: ScanningMode = .SCAN_LIVE

//set the object with Qrscanner
qrScannerHelper = QRScannerHelper.init(scanningMode: selectedMode,
bidScannerView: _viewQRScan, kQRScanResponseDelegate: self)

//start the Qrscanner
qrScannerHelper?.startQRScanning()
```

QRScanResponseDelegates

Extend your controller with this delegate to get a response from the QRscanner.

```
extension yourViewController: QRScanResponseDelegate {
    func onQRScanResult(qrCodeData: String?) {
        if qrCodeData == nil {
              //your code here
        }
        qrScannerHelper?.stopQRScanning()
   }
 }
```

User Authentication With QR Code or Push notification

Decode the string received from QRCode or push notification, and store it in an object pass that model value to the Method.

```
BlockIDSDK.sharedInstance.authenticateUser(sessionId:
datamodel.session, creds: datamodel.creds, dictScopes: [String:Any]
, lat: locationLatitude, lon: locationLongitude, origin:
datamodel.origin, userId: datamodel.userId) { (status, sessionId,
error) in

              //your code here
 }
```

User Consent

The user consent screen represents the QRscopes asked while scanning QR code. The qrCodeData string received after scanning the QR code is a base64 encoded string. This has to be converted to a JSON object after decoding. The following code snippet shows the QRScopes to be present on the UserConsent screen.

```
guard let decodedData = Data(base64Encoded: data) else {
    //invalid qr code
    return
}

// decoded string
let decodedString = String(data: decodedData, encoding: .utf8)!

//convert json string to object
let qrModel = jsonStringToObject(json: decodedString) as AuthQRModel?

//get the scope attribute
let scopesAttributes =
BlockIDSDK.sharedInstance.getScopesAttributesDic(scopes:
qrModel.scopes, creds: qrModel.creds, origin: qrModel.getBidOrigin()
, userId: selectedUserID)
```

Offline Authentication

Offline authentication means, get the user logged into your windows when your phone is in offline mode. For making this feature available, you should first have to link a user with DID and the user should be logged in once with windows in online mode.

To get the count of linked users with the application, use the below method.

```
BIDSDKHelper.shared.getLinkedUserAccounts()
```

To check if the offline is supported to the current user

```
BIDSDKHelper.shared.getCurrentUserAccount()!.isOfflineAuthSupported()
```

Generate a QR Code offline Authentication use below steps

- Set the UIView to QRGenerator view

```
@IBOutlet weak private var _qrGenerator: QRGenerator!
```

- Set below variables in your viewcontroller

```
private var timer: Timer!
private var index: Int = 0
private var result: [String] = []
private let packageSize = 250
private let timeInterval = 200
```

Set the viewWillAppear method

```
override func viewWillAppear(_ animated: Bool) {
        startQRGeneration
()
        NotificationCenter.default.addObserver(self, selector:
#selector(setUIForAddAccountButton), name:
NSNotification.Name(rawValue: "update_current_account"), object: nil)
}
```

Set viewWillDisappear method

```
override func viewWillDisappear(_ animated: Bool) {
        NotificationCenter.default.removeObserver(self)
        onFinish()
}
```

Other methods

```
private func onFinish(){
        self.resetQRGeneration()
        self.onFinishCallback!(self)
}

@objc private func setUIForAddAccountButton() {
        let currentUserAccount =
BIDSDKHelper.shared.getCurrentUserAccount()
        let linkedUserAccounts =
BIDSDKHelper.shared.getLinkedUserAccounts()
        if linkedUserAccounts.count == 0 {
            //your add account code here
        } else {
            // your current user code here
        }
        DispatchQueue.main.async {
            self.startQRGeneration()
        }
}

private func resetQRGeneration() {
        index = 0
        result = []
        if timer != nil {
            timer.invalidate()
            timer = nil
        }
}

private func startQRGeneration() {
        self.resetQRGeneration()
        self.actionQRGenerate()
}

@objc private func updateQRImage() {
        if self.result.count == self.index {
            self.index = 0
        }
        _qrGenerator.generateCode(self.result[self.index])
        self.index += 1
}


private func actionQRGenerate() {
        guard let currentUserAccount =
BIDSDKHelper.shared.getCurrentUserAccount() else {
            return self.onFinish()
        }

        guard currentUserAccount.isOfflineAuthSupported() else {
            return self.onFinish()
        }

        guard let offlinePayload =
BlockIDSDK.sharedInstance.getOfflineAuthPayload(bidLinkedAccount:
currentUserAccount) else {
            return self.onFinish()
        }

        guard timer == nil else {
            return
        }

        result = QRSplitDataPacket.split(offlinePayload:
offlinePayload, length: packageSize)
        let interval = TimeInterval(Float(timeInterval)/1000)
        self.timer = Timer.scheduledTimer(timeInterval: interval,
target: self, selector: #selector(updateQRImage), userInfo: nil,
repeats: true)
}
```

Restore

If a user wants to restore his wallet on another device, he needs to provide 12 mnemonic phrases which he has saved while registration. These phrases then will be used to get the wallet back and restore all digital assets.

To get the wallet using mnemonic phrases, use the below method

```
let walletReady =
BlockIDSDK.sharedInstance.generateWalletForRestore(mnemonics:
mnemonicPhrases)
```

Request Parameters

Params	Description
mnemonics	Type of String Space-separated 12 mnemonic phrases.

To Set restore mode on. You have to call this method before restoration begins

```
BlockIDSDK.sharedInstance.setRestoreMode()
```

To set the tenant details after generating the wallet and setting restore mode on. This function sets the tenant details

```
BlockIDSDK.sharedInstance.registerTenant(tenant: bidTenant) {(status,
error, tenant) in
    //your code here
}
```

Request Parameters

Params	Description
tenant	Type of BIDTenant

Response Parameters

Params	Description
status	Boolean value.
error	An object of ErrorResponse containing ● code: Integer ● message: String
tenant	Type of BIDTenant

To Fetch the digital assets for successful restoration Use the below methods

```
BlockIDSDK.sharedInstance.restoreUserDataFromWallet()
         { (status, error) in
      // your code here
}
```

To save the restoration data into the device, use the below method

```
BlockIDSDK.sharedInstance.commitTempData()
```

To reset the restore wallet, use the below method

```
BlockIDSDK.sharedInstance.resetRestorationData()
```

Security Check

BlockID SDK provides three types of security checks for the application.

Check DeviceAuth

Use the below method to get the authentication type and if this authentication type is enabled on a device.

```
let deviceAuth = BlockIDSDK.sharedInstance.getDeviceAuth()
```

Response Parameters

Params	Description
deviceAuth	Return type DeviceAuth isEnabled - returns if device auth is enabled. authType - value of type BiometricType .none .touchID .faceID BiometricTypeString - return string / message

Check Device Security

To check if a device is jailbroken, use the following method. It returns true or false.

```
BlockIDSDK.sharedInstance.checkDeviceTrust()
```

Check Screen Lock enabled

To check if the screen lock for the device is enabled, use the methods below. It returns true or false.

```
BlockIDSDK.sharedInstance.isScreenLockEnabled()
```

FIDO2 Capability

The below methods in the BlockID SDK enable the capability to register and authenticate FIDO2 security keys.

Register Key

The application must call the below method to perform registration of the FIDO2 security key.

```
BlockIDSDK.sharedInstance.registerFIDOKey(userName: String, tenantDNS: String,
communityName: String, completion: @escaping Fido2Callback) {
        //
}
```

Request Parameters

Params	Description
userName	String
tenantDNS	String
communityName	String
completion	Fifo2Callback

Response Parameters

Params	Description
status	Bool
error	An object of ErrorResponse containing ● code: Integer ● message: String

Authenticate Key

The application must call the below method to perform authentication of the FIDO2 security key.

```
BlockIDSDK.sharedInstance.authenticateFIDOKey(userName: String, tenantDNS: String,
communityName: String, completion: @escaping Fido2Callback) {
        //
}
```

Request Parameters

Params	Description
userName	String
tenantDNS	String
communityName	String
completion	Fifo2Callback

Response Parameters

Params	Description
status	Bool
error	An object of ErrorResponse containing ● code: Integer ● message: String

Error Codes

This section provides a list of all error codes that BlockID SDK provides.

Error Keys	Code	Message
kTenantRegisterFailed	1001	Tenant Registration Failed
kLiveIDMismatch	401	LiveID did not match
kPPDLNotMatch	410	Passport and Driver license data don't match
kDocumentPhotoComparisionFailed	411	Your photo on the document does not match with LiveID enrolled on this device
kDLBackFrontDataNotMatch	413	Driver License Front and Back don't match
kDocumentAboutToExpire	414	Document is already enrolled
kDocumentExpired	415	The document you are trying to enroll in has already expired.
kDocumentEnrolled	416	Document is already enrolled
kDocumentPhotoNotFound	417	Document photo not found
kNoMigrationNeeded	418	No migration needed
kPassportNationalIDNotMatch	419	Passport and NationalID data don't match
kNationalIdDLNotMatch	420	NationalID and Driver license data don't match
kLiveIDMandatory	421	LiveID is mandatory

K_Id_Is_Mandatory	422	Document Id is mandatory
K_Type_Is_Mandatory	423	Document type is mandatory
K_Category_Is_Mandatory	424	Document category is mandatory
K_ProofedBy_Is_Mandatory	425	Document proofed by is mandatory
K_LiveId_Not_Unenrolled	426	LiveID can't be unenrolled
K_Invalid_Category	427	Document category is invalid
K_Document_Not_Exist	428	Document does not exist
K_FirstName_Is_Mandatory	429	Document first name is mandatory
K_LastName_Is_Mandatory	430	Document last name is mandatory
K_DateOfBirth_Is_Mandatory	431	Document date of birth is mandatory
K_DateOfExpiry_Is_Mandatory	432	Document date of expiry is mandatory
K_Face_Is_Mandatory	433	Document face is mandatory
K_Image_Is_Mandatory	434	Document image is mandatory
K_LiveId_Enrolled	435	LiveID is already enrolled
K_LiveId_Wrong_Enrollment	436	LiveID can only be enrolled using setLiveID and registerDocument(with LiveID) methods

K_SSN_Mandatory	438	SSN is mandatory
kPPAboutToExpire	100001	Document is already enrolled
kPPExpired	100002	Expired passport
kInvalidPPBioData	100004	Passport Bio-Data not scanned properly
kInvalidPPEChipData	100005	Invalid E-Passport Chip Data
kPPNotMatchWithDL	100006	Passport and Driver license data do not match.
kPPNotMatchWithNID	100007	Passport and National ID data do not match.
kPPFaceNotMatchWithLiveID	100008	Your passport photo does not match with your profile photo.
kPPRFIDTimeout	100009	E-Passport Chip scan timeout
kPPPhotoNotFound	100010	Photo not found on passport
kPPRFIDUserCancelled	100011	E-Passport Chip scan was cancelled by user
kMagicLinkExpired	1410	Link expired
kMagicLinkCodeRedeemed	1429	Code already redeemed
kLicenseyKeyExpired	1003	License Key Expired
kWalletCreationFailed	1004	Wallet creation failed

kUserIdAlreadyExists	1005	UserId already exists
kSomethingWentWrong	1007	Something went wrong! Please try again!
kUnauthorizedAccess	1111	Unauthorized access
kLicenseyKeyNotEnabled	1112	License key is not enabled for this module
kPublicKeyRequired	1200	Public key required
kEncryption	1201	Something went wrong with encrypting information
kDecryption	1202	Something went wrong with decrypting information
kSomeProblemWhileFaceFinding	1008	Camera sensor is blocked.
kDocumentDataMandatory	412000	Document data is mandatory
kInvalidDL	412001	Invalid Driver License
kInvalidPP	412002	Invalid Passport
kInvalidNID	412003	Invalid National ID
kDocumentVerificationFailed	412004	Scanned document verification failed
kInvalidBase64Image	412005	Invalid base64 image

kFaceLivenessCheckFailed	412006	LiveID Liveness check failed
FIDO2 Capability		
K_RESERVED	413001	Reserved for future use
K_USERNAME_IS_MANDATORY	413002	Username is mandatory
K_TENANT_DNS_IS_MANDATORY	413003	Tenant DNS is mandatory
K_COMMUNITY_NAME_IS_MANDATORY	413004	Community name is mandatory
K_CANCELED_PROCESS	413005	The process is terminated by user
K_SESSION_EXPIRED	413006	Session has expired