

Android Mobile ID SDK Integration Guide

Overview

The **IDEMIA Mobile ID SDK** for Android is targeted towards Relying Parties who want to use the **Mobile ID App** within their existing mobile apps. The app leverages native iOS capabilities to achieve a higher level of security and the best user experience for online and offline access. It also accesses the operating system libraries and frameworks for Android, as well as **IDEMIA** and third-party SDKs.

The SDK allows you to integrate the **IDEMIA Proof** technologies into 3rd party iOS mobile applications. The enrollment APIs provide biometrics capabilities to perform facial and documents recognition.

The main enrollment use cases are:

- User registration
- User authorization
- User authentication

The enrollment functionalities consist of:

- Starting the enrollment process (with UI) and monitoring the enrollment status and results
- Cancelling the current enrollment process
- Getting the issuance data (user attributes and media)
- Unenrolling the user

Key Features

The main features of the SDK include:

- Enrollment and unenrollment (with UI)
- Collecting the enrolled user's attributes and media – API only
- In-person identity verification and data sharing (ISO 18013-5) – API only
- Online identity verification and data sharing – API only

Refer to [Release Notes](#) to see the list of improvements and fixed issues

Prerequisites

Required Skills

The integration tasks must be performed by developers with knowledge of:

- Android Studio
- Kotlin

Note: Java integration is technically possible, but it requires deep knowledge about Kotlin coroutines. Kotlin is recommended.

- Android SDK
 - Maven/Gradle dependency management
 - Kotlin Coroutines
-

Required Applications

- Android Minimum SDK 23
 - AndroidX support
-

Required Resources

Integration can be performed on a Windows/Linux/Mac OS. The tools required to perform integration are shown:

- Android Studio
- Android Device

Note: Due to fact that the SDK doesn't support x86/x64 architecture, using Android Emulators is not possible.

Required Licenses

All information about licenses configuration should be provided by your manager as a result of business process. There are two required licenses for Mobile ID SDK:

- Capture SDK license
- Document scanner license

Licenses are validated during application runtime and incorrect configuration of either license will cause application crash.

Environment Preparation

Setup gradle.properties

Setup the follow properties in the `gradle.properties` file as shown:

```
# enable androidX
android.useAndroidX=true
android.enableJetifier=true

# avoid build exception: GC overhead limit exceeded
org.gradle.jvmargs=-Xmx4096m -XX:MaxPermSize=1024m

# setup Idemia repositories credentials
idemiaRepositoryUserName=email@company.com // update this value
idemiaRepositoryPassword=password_hash // update this value
```

Note: You should receive both the username and password from your manager, along with the licenses and MID SDK configuration file.

Setup Library Repositories

Setup the library repositories in the root `build.gradle` file as shown:

```
repositories {
    maven {
        url "https://mi-artifactory.otlabs.fr/artifactory/mid-release-local"
```

```

        credentials {
            username idemiaRepositoryUserName
            password idemiaRepositoryPassword
        }
    }
    maven {
        url 'https://mi-artifactory.otlabs.fr/artifactory/smartsdk-android-local'
        credentials {
            username idemiaRepositoryUserName
            password idemiaRepositoryPassword
        }
    }
    maven {
        url 'https://mi-artifactory.otlabs.fr/artifactory/remote'
        credentials {
            username idemiaRepositoryUserName
            password idemiaRepositoryPassword
        }
    }
}
}
}

```

Note: In the case of any libraries with resolving `gradle.properties` file dependencies, follow the configuration in the sample app project.

Setup Dependencies

Add the required SDK dependencies in the application `build.gradle` file as shown:

```

interface Versions {
    def coroutines = '1.2.1'
    def smartSdkUiExtensions = '0.1.11'
    def midSdkApi = '...'
}

dependencies {
    // sdk api library
    implementation "com.idemia.mid.sdk:sdk-api:${Versions.midSdkApi}"
    // and also optional ui extensions if face capture plugin is used
    implementation "com.idemia.smartsdk:ui-
extensions:${Versions.SmartSdkUiExtensions}@aar"

    // Kotlin Coroutines
    implementation "org.jetbrains.kotlin:kotlinx-coroutines-
core:${Versions.coroutines}"
    implementation "org.jetbrains.kotlin:kotlinx-coroutines-
android:${Versions.coroutines}"
}

```

Project Configuration

Permissions

Most of the required user permissions are handled by SDK during the enrollment process; however, the `camera` permission needs to be handled before the start enrollment process as shown:

```
const val cameraRequestCode = 123

fun startEnrollment() = GlobalScope.launch {
    if (ContextCompat.checkSelfPermission(this, Manifest.permission.CAMERA) ==
PackageManager.PERMISSION_GRANTED) {
        mobileID.enrollment.start(MobileIdEnrollmentListener(applicationContext))
    } else {
        requestPermissions(arrayOf(Manifest.permission.CAMERA), cameraRequestCode)
    }
}

override fun onRequestPermissionsResult(requestCode: Int, permissions: Array<out
String>, grantResults: IntArray) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults)
    if(requestCode == cameraRequestCode) {
        startEnrollment()
    }
}
```

Native Libraries

The current **Mobile ID SDK** distribution version provides the `armeabi-v7a` and `arm64-v8a` native libraries.

You can filter out x86 and other architectures in the `build.gradle` file as shown:

```
android {
    defaultConfig {
        ndk.abiFilters 'armeabi-v7a', 'arm64-v8a'
    }
}
```

Other Required Settings

The **Mobile ID SDK** requires you to enable the data binding feature and set the code compatibility to version 1.8.

You must set this up in the `build.gradle` file as shown:

```
android {
    dataBinding {
        enabled = true
    }

    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }
}
```

Android App Bundles and Application Splitting

The Mobile ID SDK fully supports the Android App Bundle and the application split feature. This allows reduction of the final artifact size.

Note: To learn more about the app bundle, refer to the Google Android Studio site at: <https://developer.android.com/guide/app-bundle>.

Phone Binding

Phone binding plugins requires definition of the OTP Host. This configuration allows the **Mobile ID SDK** to operate with deeplinks using OTP code.

In order to setup the OTP host , add the expected value to your `strings.xml` file as shown:

```
<string name="otp_host" translatable="false">mid-dev.idemia.com</string>
```

Additionally, if you want to allow the application to read links from SMS automatically, you need to define the application public key hash by providing it to the backend team to setup as suffix of the SMS message template.

Note: To learn more about SMS verification, refer to the Google Developers site at: <https://developers.google.com/identity/sms-retriever/verify>

API Overview

The `MobileId` class from the `sdk-api` package is the main entry point for the **Mobile ID SDK**. Below you can find a list of fields that are accessible through `MobileId` with a functionality description:

- **Enrollment** - Allows you to control the enrollment process.
 - **Issuance Data Provider** - Provides access to user data gathered during enrollment process.
 - **Notifications** - Fetch the notifications and handle login requests.
 - **ISO-18013** - Allows you to process data sharing with other devices base on the ISO-18013 standard.
 - **Register/Deregister Listeners** - Observes important events like "issuance package updated".
-

SDK Configuration

The configuration delivered to you contains everything that you need to start working with the SDK API.

A sample configuration is shown:

```
{
  "version": "xx.xx",
  "ipv": {
    "apiKey": "<value>",
    "baseUrl": "<uri_value>",
```

```

        "businessProcess": "<value>",
        "otpHost": "<uri_value>",
        "messageLevelEncryption": true,
        "registrationID": "<value>",
        "audienceID": "<value>",
        "serviceProvider": "<value>",
        "clientVersion": "<value>",
        "businessId": "<value>",
        "loa": <int_value>,
        "source": "<value>",
        "action": "<value>"
    },
    "issuance": {
        "issuanceServiceKey": {
            "base64Encoded": "base64value==",
            "kid": "kid_value"
        }
    },
    "gluu": {
        "baseUrl": "<uri_value>"
    },
    "licenses": {
        "documentScanner": "base64value==",
        "lkms": {
            "serverUrl": "<uri_value>",
            "apiKey": "base64value==",
            "profileId": "<value>",
            "version": <int_value>
        }
    },
    "appInfo": {
        "contact": {
            "email": "<email>",
            "phone": "<phone_number>"
        },
        "url": {
            "about": "<url>",
            "faq": "<url>",
            "help": "<url>",
            "privacyPolicy": "<url>",
            "termsAndConditions": "<url>"
        }
    }
}

```

Configuration is read during the start of the application, and fills in the mandatory `MobileIDConfiguration` component to initialize **MobileID**.

The configuration file can be used to initialize **Mobile ID SDK** as shown:

```

import android.app.Application
import android.content.Context
import com.idemia.mobileid.sdk.MobileIdSdk

```

```
import com.idemia.mobileid.sdk.MobileIdSdkConfiguration
import com.idemia.mobileid.sdk.Enrollment

class MobileIDIntegration(app: Application) {
    private val context: Context = app.applicationContext
    private val enrollmentListener = object : Enrollment.Listener { /* TODO: implement
*/ }
    private val configuration = deserialize<MobileIDConfiguration>
(readFile("assets/mid_sdk.config"))

val mobileId = MobileID(app, configuration)

fun start() {
    mobileId.enrollment.start(enrollmentListener)
}

private fun deserialize(json: String) : MobileIdSdkConfiguration {
    // TODO: implement, just an example
}

private fun readFile(path: String) {
    // TODO: implement, just an example
}

}
```

SDK Application Theming for UI

By default the SDK uses predefined values for strings and color theme in the UI design. You can define your own values by overriding the default color strings.

SDK String Values for UI

The SDK sometimes refers examples to your application name.

You can setup your application string values in the `strings.xml` file as shown:

```
<resources>
    <string name="app_name">Your applicaiton name</string>
    <string name="app_name_for_icon">Short app name</string>
</resources>
```

SDK Color Theme for UI

The second step is to align the **Mobile ID SDK** views to your application color theme.

You can override the predefined SDK colors in your `colors.xml` file as shown:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="primary_dark">#3C0189</color>
    <color name="primary_medium">#430099</color>
```

```

<color name="primary_light">#8742E1</color>

<color name="secondary_dark">#430099</color>
<color name="secondary_medium">#8742E1</color>
<color name="secondary_light">#33430099</color>

<color name="gray_dark">#101010</color>
<color name="gray_medium_dark">#4A4A4A</color>
<color name="gray_medium">#9b9b9b</color>
<color name="gray_light">#b3babe</color>

<color name="accent_success">#429400</color>
<color name="accent_alert">#F68D0C</color>
<color name="accent_error">#D0021B</color>

<color name="background">#FFFFFF</color>

<color name="gradient_start">@color/primary_light</color>
<color name="gradient_end">@color/primary_medium</color>
</resources>

```


*Note: This step is **optional**. The SDK has encoded default values for each color.*

Example Screenshot with Color Keys

An example screenshot with the most important color keys you can override is shown in the diagram.

Sample MobileID Scan DL Screen

A sample UI screen to scan the user's drivers license is shown.

 image-20210114170711094

Sample MobileID Screen using SDK Colors

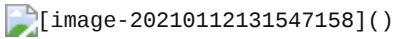
A sample Android UI screen that was designed using the SDK color strings is shown.

)

Enrollment

The enrollment process is shown in the UML diagram.

NOTE: This diagram does not include the internal (predefined) enrollment steps implementation.



Start or Restart Enrollment

To begin the enrollment process you must first prepare the enrollment parameters as shown:

```
class Config(val listener: Listener,
             val enrollmentType: Type,
             val customSteps: Set<CustomStep> = emptySet())
```

Enrollment Type

Enrollment can be started with either the `Remote` or `InPerson` mode as shown:

```
sealed class Type {
    class Remote(val businessProcess: String) : Type()
    class InPerson(val inPersonQrCode: InPersonQrCode) : Type()
}
```

Enrollment Listener

Enrollment can be finished with one of the possible scenarios shown:

- **Success** - Enrollment has been finished successfully. In this case you can access the enrollment data through `mobileId.issuanceDataProvider`.
- **Error** - Enrollment encountered an error which couldn't be handled internally by the SDK.
- **Canceled** - Enrollment has been canceled by the user.

```
class MobileIDEnrollmentListener(private val context: Context) : Enrollment.Listener {

    override fun onCancel() {
        // enrollment canceled by user
        // suggested action: navigate to last screen before enrollment start
    }

    override fun onSuccess() {
        // enrollment finished successfully
        // suggested action: navigate to your main application activity
    }

    override fun onError(error: Error) {
        // enrollment failure
        // suggested action: show error, you can use ErrorDisplayActivity delivered by
        SDK
        // -> ErrorDisplayActivity.show(context, error.value, true)
    }
}
```

Enrollment Status

You can check the current enrollment status with the `enrollmentStatus()` method from the `MobileId` object as shown:

```
class ApplicationStartupActivity : AppCompatActivity() {

    override fun onResume() {
        GlobalScope.launch {
            when(mobileId.enrollmentStatus()) {
                EnrollmentStatus.NOT_STARTED -> startEnrollmentIntroductionScreen()
                // REGISTRATION_STARTED means that registration is in progress, just
continue
                EnrollmentStatus.REGISTRATION_STARTED ->
sdk.enrollment.start(enrollmentListener)
                EnrollmentStatus.DONE -> navigateToMainActivity()
            }
        }
    }
}
```

- **NOT_STARTED** - Enrollment has not been started yet. It was cancelled or user performed an un-enrollment procedure
- **IN_PROGRESS** - enrollment was interrupted but can be continued from given step.
- **DONE** - enrollment finished successfully

Custom Enrollment Steps

Overview


This feature enables an integrator to provide their own implementation of an enrollment process step handler (e.g., capturing a selfie for face matching). It can be used to override an implementation provided by the SDK, and for steps not handled by the SDK (e.g., custom additional attributes collection).

The custom enrollment process is shown:

1. The integrator will be notified when the enrollment process reaches a step provided in the implementation. This can be handled, for example, by displaying a tutorial, capturing a photo, or showing a summary. When the data is available, the integrator must send out collected data (e.g., a selfie image).
2. After processing the data (by IPV), the integrator will be notified whether the submission was accepted or rejected.
3. In case of a fatal error, an enrollment failure callback will be executed for the SDK implementation of enrollment steps.
4. After receiving submission callback, the integrator can handle its result (e.g., display a success/error screen). As soon the step handling is done, the integrator must notify the SDK that the enrollment process can proceed with the next steps in the process.

Enrollment Activity Diagram

The custom step enrollment process is shown in the UML diagram.

 image-20210112132125842

API Enrollment Interfaces

There are two basic interfaces (in the `enrollment` namespace) that integrators should use to provide their own implementation of an enrollment step as shown in the sections below.

CustomStep Handler

The `Navigator` interface will provide a way to display screen (`Activity`) or handle it other way:

```
interface CustomStep {
    val key: String
    var priority: Int
    fun onStep(navigator: Navigator)
}
```

Custom Submission Request URL Path and Body

```
interface Submission {
    // path after `IPVConfiguration.baseURL`, eg.
    // transactions/{transactionId}/iddcoument
    val path: String
    // request payload which will be serialized to JSON, eg. a Map<String, String>
    val body: Any
}
```

MobileID.enrollment Changes

There are also two changes in the `MobileId.enrollment` object as shown:

- `start()` method signature has changed to:

```
suspend fun start(config: Config)
```

Implementations of `CustomStep` protocol can be provided in the `customSteps` field of the config object:

```
class Config(val listener: Listener,
             val enrollmentType: Type,
             val customSteps: Set<CustomStep> = emptySet())
```

- new asynchronous method `submitData` (`Submission`):

```
suspend fun submitData(submission: Submission) : Submission.Result
```

The `Result` is shown:

```
sealed class Result {
    class Success(private val action: () -> Unit) : Result() {
        fun goNext() = action()
    }
}
```

```
class Failure(val error: Exception) : Result()
}
```

**Note: The `goNext()` method must be called by the integrator when the enrollment is ready to proceed to the next step.*

Sample Enrollment Usage

1. Create the `PHONE_OTP` enrollment step handler as shown:

```
class CustomOtpCodeStep : CustomStep {

    override val key = "PHONE_OTP"

    override fun onStep(navigator: Navigator) {
        navigator.navigate(CustomOtpActivity::class.java)
    }
}
```

2. Start enrollment with `CustomStep` as shown:

```
mobileID.enrollment.start(Enrollment.Config(
    listener = enrollmentListener,
    customSteps = listOf(CustomOtpCodeStep())
))
```

3. Submit the data when it is ready to be sent:

```
val submission =
    OtpSubmission(PhoneNumber(CustomPhoneNumberActivity.phoneNumberValue), OTP(otpCode))
    when (val result = mobileID.enrollment.submit(submission)) {
        is Enrollment.Submission.Result.Success -> result.goNext()
        is Enrollment.Submission.Result.Failure -> ErrorActivity.show(this, title
= result.error.message ?: "")
    }
```

4. If there is no predefined submission, you can implement it yourself as shown:

```
class OtpSubmission(phoneNumber: PhoneNumber, otp: OTP) : Enrollment.Submission {
    override val path = "transactions/phone-binding"
    override val body = mapOf(
        IDENTIFIER to phoneNumber.number,
        "otpValue" to otp.code
    )
}
```

Note: The step shown above is optional.

Identity and Enrollment Transaction Data Access

Overview

This feature provides the integrator with access to identity and transaction data during the enrollment process. The `Identity` entity contains verified end-user data that's available at the moment (eg. data extracted from scanned document).

Transaction data contains various user data, set by the backend, which may be necessary to the integrator (e.g., captured email, phone number, etc.).

API Enrollment Class Members

The `Enrollment` class contains two new members, `transactionData` and `fetchidentity()`, as shown:

```
interface Enrollment {
    val transactionData: TransactionData?
    fun fetchIdentity() : Identity
}
```

Transaction Data

The `TranasctionData` property of `Enrollment` contains data that was already submitted during the current transaction as shown:

```
interface TranasctionData {
    val id: String
    val phone: String?
    val email: String?
}
```

Fetch Identity

The suspend `fetchIdentity()` method performs a call to IPV's to get the identity endpoint and returns the response as shown:

```
data class Identity(
    val idDocuments: Collection<IdDocument>
)
```

Issuance Data Provider

The SDK allows the retrieval of trusted data that was obtained from the issuance service during enrollment.

The data is accessible through an instance of the `IssuanceDataProvider` class, which is a property of the `MobileId` class. The `IssuanceDataProvider` methods should only be called after a successful user enrollment, otherwise an error will be thrown.

A summary of the user data is shown:

Data	Description	Method
<i>Attributes</i>	A dictionary containing information such as the user's first name, last name, gender, address, document status, etc.	<code>fun attributes(): Map<String, String></code>

<i>Media</i>	Visual elements such as portrait, signature, face animation and a template used for rendering the driver license. Related assets (e.g., frames of face animation) are stored in arrays and grouped by the type.	fun media(): Map<String, Array<ByteArray>>
--------------	---	--

Notifications

Overview

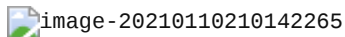
The Notifications API provides features related to the notifications service as shown:

- Online attribute sharing
- Online authentication
- SOR initiated attributes update
- Ad-hoc messages receiving

This includes providing access to pending and historical notifications, as well as push notifications handling.

Notifications Activity Diagram

The way the status changes, based on the actions taken by the user, is presented in the UML diagram:



Notifications Integration

The **NEW** notifications should be fetched at the beginning of the application, after push notification and on connection are received. The Integrator can ignore the fetched notifications, but they are needed to allow the SDK to work properly with Listeners.

API Sample Code

The sample code for `Notifications` is shown:

```
interface Notifications {

    suspend fun handle(loginRequest: Login.Request): Login.Session
    suspend fun fetch(filters: Filters): Collection<Notification>

    val push: PushNotifications
}
```

Login Request

You can create the Login request with `qrCode` or `deeplink` as shown:

```
val qrCode = QrCode( SCANNED_QR_CODE_VALUE )
val loginRequest = Login.Request(qrCode)
```

```
val deeplink = Deeplink( DEEPLINK_URL )
val loginRequest = Login.Request(deeplink)
```

Next, you must handle it with the `Notifications` API:

```
suspend fun onQrCodeScanned(qrCodeValue: String) : Notifications.Notification {

    val qrCode = QrCode(qrCodeValue)
    val loginRequest = Login.Request(qrCode)

    return notifications.handle(loginRequest).pull()
}
```

```
val notification = onQrCodeScanned(qrCodeValue)
notification.acknowledge() // change notification status to READ
if(notification is Acceptable) {
    notification.accept() // or notification.decline()
}
```

Push Notifications

On the *Google Firebase* platform, the new token callback integrator should also pass that value to the SDK as shown:

```
mobileId.notifications.push.tokenReceived(TOKEN_VALUE)
```

ISO-18013

Overview

This feature provides a simple API to implement the ISO-18013 standard, without requiring deep knowledge about the ISO specification.

The API hides all implementation details like bluetooth connection. It also provides simple methods that guide the integrator through all the process steps.

API Outline

The main **Mobile ID SDK** object contains the ISO18013 object that allows the integration to create a new ISO session.

Creation session prepares the device engagement data, and generates the session encryption keys as shown:

```
class MobileID {
    // ...
    val iso18013: ISO18013
}

interface ISO18013 {
    fun createSession(): Session
}
```

Next, the integrator works with the `Session` object. The device engagement data is contained in an array, and the possible methods you can use are `QrCode` or `NFC`.

The integrator has to share this data, for example, read the QR code value and display it in their own application.

The integrator also has to start the process by passing the expected `Listener` :

```
sealed class DeviceEngagementOption {    // currently only QrCode is supported!
    class QrCode(val payload: String) : DeviceEngagementOption()
}

interface Session {
    interface Listener {
        fun onConnectionEstablished()           // BLE connected
        fun onRequestReceived(request: Request) // can be called multiple times
        fun onSessionFinishedSuccessfully()      // session end, all responses sent
                                                successfully
        fun onError(e: IsoException)
    }

    val deviceEngagementOptions: Array<DeviceEngagementOption> // array of possible
                                                                methods, eg. QR code
    suspend fun start(listener: Listener)
}
```

After starting the process, the integrator waits for callbacks on the passed listener.

The expected call in the process is `onRequestReceived` , where the parameter `request` implements the interface as shown:

```
sealed class Response {
    object Accept : Response()
    object Decline : Response()
}

data class DocumentRequest(val key: String)

interface Request {
    val version: String
    val requested: Collection<DocumentRequest>

    suspend fun sendResponse(response: Response): Transfer
}
```

In the most common flow, the integrator should display to the user their list of requested attributes, and then based on the user's decision, send a response.

The `sendResponse` method returns the object that is implementing the `Transfer` interface as shown:

```
interface Transfer {
    val progress: TransferProgress
}
```



```

suspend fun start(listener: Listener)

interface Listener {
    fun onProgressChanged(progress: TransferProgress)
    fun onTransferFailure(e: IsoException)
}

interface TransferProgress {
    val itemsSent: Int
    val itemsTotal: Int

    val isCompleted: Boolean
    get() = itemsSent == itemsTotal

    val fractionCompleted: Double
    get() = Double(itemsSent) / Double(itemsTotal)
}


```

Next, the integrator has to call `start(listener)` and begin data transferring and follow the progress.

If there is there is any other request made during the current session, the `onRequestReceived` method will be called again. If there is not and all data transfers finish successfully, `onConnectionClosedSuccessfully` will be called.

ISO API Activity Diagram

The ISO activity for APIs is shown in the UML diagram below:

 image-20210112131243195

ISO Exceptions

Some errors that could occur during sharing sessions are shown:

Exception	Explanation
CommunicationException	error of the underlying medium
<code>Iso18013PackageNotFoundException</code>	ISO package was not returned during issuance enrollment step or during attribute update
ReceivedErrorException	error received with session data
SessionEncryptionFailureException	cryptography-related error during session establishment, request decryption or response encryption
SessionEndedPrematurelyException	the reader sent an "end" state before the transfer has completed
UnparseableRequestException	The client failed to parse an incoming SessionData or Request

Regarding `sessionEndedPrematurely`, the current **Mobile ID** behavior is to report success (i.e., NOT showing this error) if at least one transfer completes successfully. Theoretically, it would make more sense to require ALL transfers to complete.

Note: The exact logic on the SDK side, and how much should be left for the integrator to be concerned with, still needs to be discussed.

Sample Integration Code

```
interface IsoView {

    // Show Device Engagement Qr Code to start communication
    fun showQrCode(qrCode: String)

    // Verifier app connected, waiting for request
    fun showConnectionEstablishedScreen()

    // Request from Verify app received
    fun showRequest(requestedItems: List<String>, requestListener: RequestListener)

    // Transfer progress methods
    fun showTransferView()
    fun showTransferProgress(progress: Int)

    fun showSuccessScreen()
    // Error occurs during process
    fun showError(error: String)

}
```

```
class IsoPresenter(iso: ISO18013, private val view: IsoView) {

    private val isoSession = iso.createSession()

    suspend fun start() {
        log.d("Start")
        val qrCode = isoSession
            .deviceEngagementData
            .first { it is ISO18013.DeviceEngagement.QrCode } as
            ISO18013.DeviceEngagement.QrCode
        view.showQrCode(qrCode.encoded)
        isoSession.start(IsoTransactionListener(view))
    }

}

private class IsoTransactionListener(private val view: IsoView) :
    ISO18013.Session.Listener {

    private val requests: MutableList<IsoRequestListener> = mutableListOf()
```

```

    override fun onConnectionEstablished() {
        log.d("On connection established")
        view.showConnectionEstablishedScreen()
    }

    override fun onRequestReceived(request: ISO18013.Request) {
        log.d("Request received")
        val requestedItems = request.requested.map { it.key }
        val listener = IsoRequestListener(view, request)
        view.showRequest(requestedItems, listener)
        requests.add(listener)
    }

    override fun onConnectionClosedSuccessfully() {
        log.d("On connection closed")
        view.showSuccessScreen()
    }

    override fun onError(e: IsoException) {
        log.e("Transaction listener - Error", e)
        view.showError(e.message ?: "")
    }
}

private class IsoRequestListener(private val view: IsoView, private val request:
ISO18013.Request) : RequestListener {

    private val transferListener = IsoTransferListener(view)

    override suspend fun onAccept() {
        log.d("Request accepted")
        view.showTransferView()
        request.sendResponse(ISO18013.Response.Accept).start(transferListener)
    }

    override suspend fun onDecline() {
        log.d("Request declined")
        view.showTransferView()
        request.sendResponse(ISO18013.Response.Decline).start(transferListener)
    }
}

private class IsoTransferListener(private val view: IsoView) :
ISO18013.Transfer.Listener {

    override fun onProgressChanged(itemsSent: Int, itemsCount: Int) {
        val progress = itemsSent * 100 / itemsCount
        log.d("Progress : $progress%")
        view.showTransferProgress(progress)
    }
}

```

```
        override fun onTransferFailure(e: IsoException) {
            log.e("Transfer listener - Error", e)
            view.showError(e.message ?: "")
        }
    }
}
```

Additional Features

Analytics

In order to improve our algorithms, by default the **Mobile ID SDK** collects telemetric data on the external server, for example, the face capture success ratio. *Note: Be advised that the SDK never sends sensitive data.*

You can disable all analytics with the one line of code shown:

```
val mobileId = MobileID(app, configurationForSDK)
mobileId.disableAnalytics()
```

Error Screen

Errors that terminate the enrollment process can be handled by the integrator using the custom UI or predefined error screens in the SDK as shown:

```
class MobileIdEnrollmentListener(private val context: Context) : Enrollment.Listener {

    override fun onError(error: ErrorCode) {
        ErrorDisplayActivity.show(context, error.value, true)
    }
}
```