
iOS Integration Guide for Mobile ID SDK

Overview

The **IDEMIA Mobile ID SDK** for iOS 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 end-user experience for both online and offline access. It also accesses the operating system libraries and frameworks for iOS, 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.

Accessing User Attributes

The user's identity attributes are downloaded from a server during the back-end enrollment process and stored in the user's phone. After the user inputs all necessary data in the UI, the issuance step occurs (there is not need for the integrator to do anything). Then the completion callback is called and the end-user enrollment completes.

These user attributes can be accessed using the [IssuanceDataProvider](#) event.

Prerequisites

Required Skills

Integration requires that developers have a working knowledge of the following:

- Xcode
 - Swift
 - iOS (minimum version is 11.0)
 - Cocoa Pods (optional)
-

Required Resources

Integration must be performed on a Mac.

The required tools are:

- Xcode 11.5
 - iOS device (preferred) or simulator
-

Required Licenses

The required licenses for Mobile ID SDK are:

- The Capture SDK license
- The Document Scanner license

Both of these licenses should have been provided to you in the [Configuration section](#).

SDK Entry Point

The `MobileID` class is the entry point for all SDK features.

These features include:

- `**Enrollment**`: the object responsible for the enrollment process.
 - `**IssuanceDataProvider**`: the object that provides user data (following a successful enrollment process).
-

SDK Plugins

The **MobileID/Core** plugin provides the `MobileID` module and these plugins containing the SDK features:

- **MobileID/EnrollmentPlugin***: Plugins responsible for the enrollment process. The list of required plugins depends on backend services configuration and it should have been provided to you alongside the [Configuration](#).
 - **MobileID/Notifications**: See [Notifications](#).
 - **MobileID/ISO18013**: See [ISO 18013-5](#).
-

Project Configuration

Step 1: Configure your project's Info.plist

Camera Access

Add the *Privacy - Camera Usage Description* (`NSCameraUsageDescription`) key and a description.

Below is an example `Info.plist` entry:

```
<key>NSCameraUsageDescription</key>
<string>Used to scan the front and back of your driver license and your face.
</string>
```

Phone Binding

Note: These steps are optional depending on the business process.

1. Add a new item to *URL types* (`CFBundleURLTypes`).
2. Create an new item and select *URL Schemes* (`CFBundleURLSchemes`)
3. Add the URL scheme that your project will use for phone binding.

The result should display similar to this:

```
<key>CFBundleURLTypes</key>
<array>
  <dict>
    <key>CFBundleURLSchemes</key>
    <array>
      <string>$YOUR_SCHEME</string>
    </array>
  </dict>
</array>
```

This value for `CFBundleURLTypes` should have been supplied to you by your provider.

Note: The phone number is binded with the application. This value differentiates between other applications on the phone. The user gets a text message with their link. The user provides their phone number because a unique value is needed for each customer.

Application Long Name

Add the `ApplicationLongName` key. This entry is necessary for several information and error screens to display correctly.

This is a **MobileID** specific entry; you will not find it in the dropdown provided by Xcode.

Step 2: Bundle server certificates

The SDK uses SSL pinning, which is mandatory (at the time of this release).

`Alamofire` facilitates the underlying process and details of pinning, including the certificates in the bundle is all that is required.

All required certificates will be provided to you along with the configuration.

Step 3: Integrate required frameworks

The SDK is distributed as a set of modules, consisting primarily of dynamic frameworks. These modules can be integrated either via CocoaPods or manually.

The SDK modules include:

- `MobileID` (1.0.0) - the SDK entry point
- `MobileIDEnrollmentPlugin*` (1.0.0)
- `MobileIDNotifications` (1.0.0) - optional
- `MobileIDISO18013` (1.0.0) - optional

The internal dependencies include:

- `openssl-mid` (1.1.102.1)
- `BiometricSDK` (= 4.23.0)
- `BiometricSDKUIFaceModeHigh` (= 2.1.0-beta4)
- `BiometricSDKUIFaceModeMedium` (= 2.1.0-beta4)
- `PPBlinkID` (5.8.0-IDEMIA)

The external dependencies include:

- `Alamofire` (5.2.2)
- `CocoaLumberjack` (3.6.2)
- `DeviceKit` (4.1.0)
- `FlagPhoneNumber` (0.8.0)
- `GzipSwift` (5.1.1)
- `KeychainAccess` (4.2.1)
- `LifetimeTracker` (1.8.0)
- `Localytics` (6.1.0)
- `lottie-ios` (3.1.8)
- `SSZipArchive` (2.2.3)

Add Mobile ID SDK Frameworks

We serve our frameworks via artifactory.

Frameworks are available at the <https://mi-artifactory.otlabs.fr/artifactory/webapp/> page.

As an integrator, you can choose to include frameworks within your project in one of two ways:

- CocoaPods integration; or
- manual integration.

CocoaPods Integration

Note: This includes the `cocoapods-art` plugin.

1. CocoaPods doesn't support authentication, so in order to use CocoaPods with Artifactory you must install the [cocoapods-art](#) plugin. Run the following command from your terminal:

```
gem install cocoapods-art
```

2. The plugin uses authentication details as specified in the standard [.netrc file](#).

```
machine mi-artifactory.otlabs.fr
login ##USERNAME##
password ##PASSWORD##
```

3. Once set, run this command to add our repository to your CocoaPods dependency management system:

```
pod repo-art add mobileidsdk "https://mi-artifactory.otlabs.fr/artifactory/api/pods/smartsdk-ios-local"
```

4. At the top of your project's `Podfile` add this command:

```
plugin 'cocoapods-art', :sources => [
  'master',                # to resolve dependencies from the master
  repository.
  'mobileidsdk'            # to resolve the MobileID dependency.
]
```

5. Add `MobileID` in your `Podfile` in one of the pod variants:

```
pod 'MobileID/EnrollmentPlugin*' # Specify all enrollment plugins you need.
Note that the *Core* module is always included.
# optional
# pod 'MobileIDNotifications'
# pod 'MobileIDISO18013'
```

6. Run the install command as usual:

```
pod install
```

If you are already using the **IDEMIA** repository and cannot resolve a dependency, try to update the specs as shown:

```
pod repo-art update mobileidsdk
```

Manual Integration

1. Download the artifacts from the Artifactory ([repository link](#)) page and unpack its contents.
2. In the Xcode project editor, select the target to add the library or framework.
3. Click **Build Phases** tab at the top of the project editor.
4. Open the **Embedded Binaries** phase. *Note: if this section is missing, you may perform the following steps under the **General** tab's **Frameworks, Libraries, and Embedded Content** section as an alternative.*
5. Click the Add button (+).
6. Click the **Add Other...** button below the list.
7. Add the following:
 - All frameworks from `MobileIDCore`, the selected `MobileIDEnrollmentPlugin`, and any other optional `MobileID*` frameworks

- o All dependent frameworks

Configuration

The configuration delivered to you contains everything required to work with the SDK API. Except for `colors` and `appInfo`, you should not change any values.

Below is a sample configuration:

```
{
  "enrollmentPlugins": [
    "<Module.Plugin>"
  ],
  "ipv": {
    "apiKey": "<value>",
    "baseUrl": "<uri_value>",
    "messageLevelEncryption": true,
    "registrationID": "<value>",
    "audienceID": "<value>",
    "serviceProvider": "<value>",
    "clientVersion": "<value>",
    "businessId": "<value>",
    "loa": 2,
    "source": "<value>",
    "action": "<value>"
  },
  "issuance": {
    "dataGroup": "application/signed-tree+v2",
    "issuanceServiceKey": {
      "base64Encoded": "<value>",
      "kid": "<value>"
    }
  },
  "authServer": {
    "gluuUrl": "<value>",
    "jwtCacheUrl": "<value>"
  },
  "licenses": {
    "documentScanner": "<value>",
    "lkms": {
      "apiKey": "<value>",
      "serverUrl": "<value>",
      "profileId": "<value>",
      "version": 3
    }
  },
  "colors" : {
    "primary": "<hex-value>",
    "secondary": "<hex-value>",

    "grayscaleDark": "<hex-value>",
    "grayscaleMediumDark": "<hex-value>",
    "grayscaleMedium": "<hex-value>",
    "grayscaleLight": "<hex-value>",

    "accentRed": "<hex-value>",
```

```

        "accentOrange": "<hex-value>",
        "accentGreen": "<hex-value>",

        "background" : "<hex-value>"
    },
    "appInfo": {
        "contact": {
            "phone": "+12 345 67 89",
            "email": "foo@bar.xyz"
        },
        "urls": {
            "about": "https://foo.bar/xyz",
            "faq": "https://foo.bar/xyz",
            "help": "https://foo.bar/xyz",
            "privacyPolicy": "https://foo.bar/xyz",
            "termsAndConditions": "https://foo.bar/xyz",
        }
    }
}

```

Colors

The SDK UI will only use the two thematic colors defined in the configuration:

- **Primary:** used for single, main elements (e.g., the navigation bar background and photo borders).
- **Secondary:** used for elements that occur multiple times on the screen (e.g., buttons and links).










Colors prefixed with **grayscale** are generally used for text (e.g., dark, medium dark, medium), borders/dividers (e.g., medium, light), and inactive elements (e.g., medium).

Colors prefixed with **accent** are used for success, error, and warning indications.

The tint color is usually set to **primary**, although exceptions exist. For example, if the background is dark, white will be used instead (e.g., for primary button text).

SDK Color Palette - UI Screen

The SDK color palette for the UI screen design is shown.

-  #263347 Primary
-  #96733B Secondary
-  #FFFFFF Background
-  #101010 Grayscale dark
-  #4A4A4A Grayscale medium dark
-  #9B9B9B Grayscale medium
-  #B3BABE Grayscale light
-  #429400 Accent green (Success/positive)
-  #F68D0C Accent orange (Pending/Alert)

Sample UI Screen with SDK Colors

A sample UI screen designed with the SDK color palette is shown.



Sample MobileID Scan DL Screen

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



Sample MobileID Capture DL Screen

A sample UI screen that takes a photo of the user's drivers license is shown.



App Info

App info is used when the SDK needs to provide information about your app to the end user (e.g., when an error occurs or the user needs support).

The `appInfo` section in the configuration is split into two sections:

- **Contact:** contains the phone number and email address for support.
- **URLs:** contains the URLs for the *More* menu options.

MobileIDConfiguration

You must provide an instance of `MobileIDConfiguration` to initialize the SDK.

The configuration is a Swift `struct` (as opposed to a JSON file) for security reasons. This configuration contains confidential data, such as the API or license keys, so it is best to avoid including it in the bundle.

In the event you *do* want or need to include it, use the JSON format. All the configuration structures conform to `Decodable`.

SKD Entry Point

Initialization

`MobileID` is the main entry point for the SDK.

Prior to using the Mobile ID SDK, it must first be initialized with a [configuration](#) as shown:

```
let configuration = MobileIDConfiguration(...)
let mobileID = try MobileID(configuration: configuration)
```

User Status

The user status can be retrieved by calling `mobileID` as shown:

```
mobileID.update { status in
    switch status {
        case .active:
            // The user is active, its data are accessible via
            `mobileID.issuanceDataProvider`.
            break
        case .enrollmentInProgress:
            // An ongoing enrollment process can be resumed by calling
            `mobileID.enrollment.start()`.
            break
        case .notEnrolled:
            // The user has not been enrolled yet, a new enrollment process can
            be started by calling `mobileID.enrollment.start()`.
            break
        case .cancelled:
            // The user profile has been cancelled, `mobileID.update()` should
            be called to check if it is reinstated.
            break
        case .updateRequired:
            // Before performing any other action, `mobileID.update()` must be
            called to perform required data updates.
            break
    }
}
```

Handling Push Notifications

A significant part of **Mobile ID** functionality relies on the SDK being able to process push notifications.

At the beginning of the enrollment flow the user will be automatically prompted for appropriate permissions, however you will need to pass the device token to the SDK as shown:

```
func application(_ application: UIApplication,
didRegisterForRemoteNotificationsWithDeviceToken deviceToken: Data) {
    mobileID.setDeviceToken(deviceToken)
}
```

Verify that the appropriate Background Modes (Background Fetch and Remote Notifications) are enabled for your app.

Screen Orientation

To ensure proper screen orientation (portrait, landscape) for some views, you need to check in `AppDelegate` to see if the current view controller implements the `Theme.SupportsRotation` protocol as shown:

```
import Theme

class AppDelegate: UIResponder, UIApplicationDelegate {
    func application(_ application: UIApplication,
supportedInterfaceOrientationsFor window: UIWindow?) ->
UIInterfaceOrientationMask {
        guard let currentViewController =
window?.rootViewController?.currentViewController as? SupportsRotation else {
            return .portrait
        }

        return currentViewController.supportedOrientations()
    }
}
```

Handling Deep Links

Depending on the business process, enrollment might involve phone binding. For a smooth phone binding to work, verify that the proper scheme is set in the [Info.plist of your project](#).

You must pass the deep links that use this scheme to the SDK using the `AppDelegate` or `SceneDelegate` class as shown:

```
class AppDelegate: UIResponder, UIApplicationDelegate {
    application(_ app: UIApplication,
        open url: URL,
        options: [UIApplication.OpenURLOptionsKey : Any] = [:]) -> Bool {
        mobileID.handleDeepLink(url: url)
    }
}
```

If everything is setup correctly, the user will receive an SMS during enrollment with a link containing a one-time password. Tapping on that link will move the user back to the app and allow the enrollment to continue.

Data Update

When a user's data is changed in the system managed by an Issuing Authority (for example the System of Records), the data stored on the device must be updated accordingly.

It is imperative that updates are not ignored, because they can also convey information that the user's document has been revoked or has expired.

There are two kinds of updates as shown:

- signed assertion-based updates, which are triggered by the [Notifications] (./notifications.html) service
- on-demand updates, which are initiated by the SDK and involve authentication using already possessed document data instead of a signed assertion

Once an update finishes successfully, the summary of the changes can be received by assigning a listener method as shown:

```
mobileID.events.onUpdateFinished = { (summary: UpdateSummary) in  
}
```

Make sure to assign that callback as early as possible, before any updates are triggered.

Signed Assertion-based Update

When a user's data is changed in the system, the **Mobile ID** backend sends out a push notification.

That notification is a cue for the app that an update should be performed as shown:

```
mobileID.update { error in  
    // check if any error occurred  
}
```

This method first checks if there is a pending update, and will finish without error if there is not. It is therefore strongly encouraged to call this method regularly; for example, when the app enters the foreground and as a part of the Background App Refresh.

Data Update on Demand

A data update can also be performed regardless of whether a notification with a signed assertion has been received; for example, when an external event happened and bypassed the **Mobile ID** backend system.

```
mobileID.forceUpdate { error in  
    // check if any error occurred  
}
```

Enrollment

The enrollment process is shown in the UML diagram.



Start or Resume Enrollment

1. Import the framework modules as shown:

```
import MobileID
import Theme
```

2. Instantiate and initialize the Mobile ID SDK (if not already done) as shown:

```
let configuration = MobileIDConfiguration(...)
let mobileID = try MobileID(configuration: configuration)
mobileID.initialize { error in
    // handle optional initialization error or proceed further
}
```

3. Create `MIDNavigationController` and make it visible as shown:

```
let navigationController = MIDNavigationController()
window.rootViewController = navigationController
```

4. Create the enrollment parameters as shown:

```
let enrollmentType: EnrollmentType = ...
let customSteps: [Enrollment.CustomStep] = ...
let parameters = Enrollment.Parameters(
    enrollmentType: enrollmentType,
    customSteps: customSteps
)
```

- Use `EnrollmentType.inPerson` to start *in-person enrollment*. Note: You will need to provide a valid QR code.
- Use `EnrollmentType.remote` to start *document-based enrollment*. Note: You will need to provide a business process identifier.

5. Start enrollment with a completion handler as shown:

```
mobileID.enrollment.start(with: parameters, in: navigationController) {
    result in
        switch result {
            case .success:
                // now you can use \Issuance Data Provider
            case .failure(let error):
                // handle the error
            case .cancelled:
                // user cancelled the enrolment
        }
}
```

Cancel Enrollment

Cancel the enrollment process as shown:

```
mobileID.enrollment.cancel()
```

Unenroll

Unenroll the user as shown:

```
mobileID.enrollment.unenroll()
```

Custom Enrollment Steps Handling

You can provide own implementation for most of enrollment steps. In order to do so, you need to:

1. Make a type conforming to `Enrollment.CustomStep` protocol for every step you need to handle by your own:

```
class AdditionalAttributesStep: CustomStep {  
    let key = "ADDITIONAL_ATTRIBUTES"  
  
    func onStep(navigator: Navigator) {  
        // TODO show your own UI, make API calls etc  
    }  
}
```

Its the integrator's responsibility to collect all required data that are needed in current enrollment step (state).

2. Provide instances of those types in `Enrollment.Parameters` 's initializer:

```
let configuration = Enrollment.Parameters(customSteps:  
[AdditionalAttributesStep()])
```

3. Use it when starting enrollment process:

```
mobileID.enrollment.start(with: configuration, ...)
```

4. When you are done collecting all required data, you need to send them to the enrollment backend services:

```
mobileID.enrollment.submit(AdditionalAttributesSubmission(["attributeKey":
"attributeValue"])) { result in
    switch result {
        case .success(let doneCallback):
            // do whatever you need after succesful submission,
            // eg. show success screen.
            // Then, when this custom step handling is done:
            doneCallback()
        case .failure(let error):
            // In case submitted data were rejected (eg. validation failed),
            // you can retry collecting the data and sumit them again.
    }
}
```

The `submit()` method first argument must conform to `Submission` interface. You can use the pre-existing types provided by the SDK or implement your own.

In case of a submission resulting in enrollment failure, the `submit()` method completion handler won't be called. Instead, the `Enrollment.start()` completion handler will be called with a failure.

Verified Identity Data Access

During the enrollment process, it is possible to access the verified identity data available at the moment.

Access the verified data extracted from the scanned document as shown:

```
mobileID.enrollment.fetchIdentity { result in
    guard let idDocumentData = try?
result.get().idDocuments?.first?.idDocumentData else {
        return
    }
    let documentNumber = idDocumentData.idDocumentNumber
    let surname = idDocumentData.personalAttributes.surname.value
}
```

Transaction Data

During the enrollment process, it is possible to access the captured email address and phone number as shown:

```
if let data = mobileID.enrollment.transactionData {
    let email = data.email
    let phoneNumber = data.phone
}
```

Running Enrollment on Simulator (Question)

For development purposes, it is possible to run enrollment on an iOS simulator. Because there is no camera in the simulator, enrollment data must be provided.

If document authentication is enabled for the front of the user's identity document:

```
#if targetEnvironment(simulator)
Enrollment.setMockSelfie(image: UIImage(named: "face")!)
#endif
```

```
#if targetEnvironment(simulator)
Enrollment.setMockDocument(front: UIImage(named: "front")!, back: UIImage(named:
"back")!)
#endif
```

If document authentication is enabled for both the front and back of the user's identity document:

```
#if targetEnvironment(simulator)
Enrollment.setMockData(documentFront: UIImage(named: "front")!, documentBack:
UIImage(named: "back")!, selfieImage: UIImage(named: "face")!)
#endif
```

The quality of the images must be sufficient to read the PDF417 barcode from the back document image, in order to compare it to the front of the document containing the owner's photo with selfie.

Issuance Data

The SDK allows for the retrieval of trusted data. This data is 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` method should only be called after a successful enrollment, otherwise an error will be thrown.

The summary of data is shown in the table below.

Data	Description	Method
<i>Attributes</i>	A dictionary containing information (e.g. first name, last name, gender, address, document status, etc)	<code>func attributes() throws -> [String: String]</code>
<i>Media</i>	Visual elements, such as the portrait, signature, face animation, and a template used for rendering the driver license. Related assets (e.g. frames of the face animation) are stored in arrays and grouped by their type.	<code>func media() throws -> [String: [Data]]</code>

Error Handling

The SDK returns errors either using `throwing` methods or the `Result` enum from the standard Swift library, depending on whether a given method is asynchronous or not.

You can handle these errors as desired, but the SDK offers a convenience method to present complete, themed error screens as shown:

```
public func showError(_ error: Error, in rootViewController:
    UINavigationController, primaryButtonAction: @escaping () -> Void)
```

The example usage is shown:

```
} catch {
    mobileID.showError(error, in: rootViewController) {
        // proceed
    }
}
```

The `rootViewController` could be the same `UINavigationController` instance that you passed to the `start` method.

Notifications

Overview

Notifications is a module of the Mobile ID SDK responsible for receiving and processing notifications produced by the Notification Service.

Some notifications are merely informational and intended to be presented to the user, while others are actionable and require user presence and consent to release personal information to a Reliant Party.

Activity Diagram

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



Usage

An instance of Notifications is available as a property of the `mobileID` class:

```
let notifications = mobileID.notifications
```

Receiving Notifications

Manual Fetching

Notifications can be fetched from backend service using `fetch(_ request:, completion:)` method. The `FetchRequest` structure allows you to set requested page number, page size and list of notification statuses to filter by. Resulting notifications are ordered descending by creation date.

```
notifications.fetch(FetchRequest(page: 0, pageSize: 10, statuses: [.new])) {
    result in
        let notifications: [NotificationCard] = try result.get()
        print(notifications)
}
```

Push Notifications

The Notification Service can send out push notifications to your application to notify it that a new Notification Card is available.

Make sure you have followed the steps outline [here](#).

Then, upon receiving a push notification, perform a fetch to check what new cards are available (there can be more than one).

```
func application(_ application: UIApplication,
                 didReceiveRemoteNotification userInfo: [AnyHashable: Any],
                 fetchCompletionHandler completionHandler: @escaping
(UIBackgroundFetchResult) -> Void) {
    notifications.fetch(FetchRequest(statuses: [.new])) { result in
        switch result {
        case .success(let notifications):
            completionHandler(notifications.isEmpty ? .noData : .newData)
        case .failure:
            completionHandler(.failed)
        }
    }
}
```

Fetching Frequency

Push notifications are supposed to provide a cue for you to perform a fetch. However, they are not 100% reliable - the user might not grant the permission, or the APNS itself might have issues.

It is therefore strongly encouraged to perform the fetch regularly, for example on app entering foreground and as a part of Background App Refresh.

Notification Actions

Actions that can be performed on a notification card are provided by its `actions` property.

The value is a dictionary of closures, where keys represent the meaning of the given closure i.e. when it should be fired.

Which actions are available at a given time depends on the origin, as well as the status, of the notification.

Currently there are three possible actions:

Action	Effect	Note
<i>Accept</i>	Grants approval for online authentication / attribute sharing.	Available only for online authentication / attribute sharing notifications.
<i>Decline</i>	Declines approval for online authentication / attribute sharing.	Available only for online authentication / attribute sharing notifications.
<i>Acknowledge</i>	Used for marking the notification as read.	Available for all notifications.

You can present different buttons depending on the contents of the `actions` dictionary, or simply check if a specific action type is included as shown:

```
if notification.actions.keys.contains(.accept) {  
    // present "Accept" button  
} else {  
    // present "OK" button  
}
```

Alternatively you can map the `actions` into buttons - this is especially useful with SwiftUI, but is outside of the scope of this guide.

```
let buttons = notification.actions.map { actionType, closure in  
    // return a button based on the action  
}
```

Calling Actions

Each action provides a callback with an optional `Error` value.

You can call a given action directly:

```
request.actions[.accept]? { error in  
}
```

Notification Status

Notification cards are inherently stateful. Their lifecycle is represented by their `status` property.

There are five possible statuses for a notification:

- new
- read
- accepted
- declined
- expired

Online Authentication and Attribute Sharing

The Notifications framework allows the user to log into claims-based applications, referred to as Relying Parties, and share their Drivers License (DL) information with them.

The flow can be initiated in a number of ways:

- Visiting the Relying Party website on a computer and scanning the QR code
- Visiting the Relying Party website on the mobile device and opening a deep link to the **Mobile ID** enabled application
- The Relying Party can also create a request by themselves, for example the Department of Revenue can send out tax return information for approval

In each case above, the user will receive a notification that requires a response. Typically, the notification will feature an expiration date that determines how much time the user has to respond.

Activity Diagram

The user-initiated flows are illustrated in the UML diagram as shown.



Analytics

The SDK offers a way to track analytics events.

If you want to receive these events, you need to implement the `Tracker` protocol as shown:

```
public protocol Tracker {
    func trackEvent(_ event: TrackableEvent)
    func trackEnrollmentStart(transactionId: String)
    func trackEnrollmentEnd()
}
```

and pass it to the `MobileID` initializer:

```
let tracker = MyTracker()
let sdk = try MobileID(configuration: configuration, tracker: tracker)
```

IDEMIA Tracker

By default, the SDK will also forward the events to an IDEMIA tracker for the purpose of improving the product.

If you want to opt out of this, simply pass `analyticsOptOut:true` when initializing the SDK:

```
let sdk = try MobileID(configuration: configuration, analyticsOptOut: true)
```

List of Events

Event	Parameters	Description
<i>Begin Tutorial</i>		Event is sent when tutorial screen in welcome activity appears.
<i>Accept Payment Screen</i>		Event is sent when user clicks "Try mobile id for free" button on payment screen.
<i>Accept Registration Terms</i>		Event is sent when "I agree to above" button on permissions screen is clicked.
<i>Registration Start</i>		Enrollment transaction is created by the backend.
<i>Registration Success</i>		The last step of the transaction completes successfully.
<i>Registration Failure</i>	<ul style="list-style-type: none"> error code error message 	Enrollment fails at any step.
<i>Registration Timeout</i>		Transaction times out.
<i>Registration Cancelled</i>		User cancels enrollment at any step.
<i>Accept Registration Terms</i>	<ul style="list-style-type: none"> time elapsed since the entering the "Let's activate your Mobile ID" screen 	User taps "Activate" button AFTER going through the enrollment.
<i>Decline Registration Terms</i>	<ul style="list-style-type: none"> time elapsed since the entering the "Let's activate your Mobile ID" screen 	User taps "Yes" on the "Are you sure you want to cancel?" prompt.
<i>Phone Binding Started</i>		User enters phone number entry screen.
<i>Phone Info (Success / Failure)</i>	<ul style="list-style-type: none"> time elapsed since the "Phone Binding Started" event error code (if available) error message (if available) 	Phone number submission to server completes
<i>Phone OTP (Success / Failure)</i>	<ul style="list-style-type: none"> # of retries time elapsed since the "Phone Info Success" event error code (if available) error message (if available) 	OTP submission to server completes. Failure means either server connectivity issues or the server rejecting the OTP.

Event	Parameters	Description
<i>(Front / Back) Scan Started</i>		User enters the horizontal scanning screen (after the tutorials).
<i>(Front / Back) Picture Scanned</i>	<ul style="list-style-type: none"> • # of retries • time elapsed since the "(Front / Back) scan started" event 	User scans a side of a document successfully.
<i>(Front / Back) Scan Failure</i>	<ul style="list-style-type: none"> • # of retries • isTimeout • error code • time elapsed since the "(Front / Back) scan started" event 	User fails an attempt to scan a side of a document.
<i>Selfie Captured</i>	<ul style="list-style-type: none"> • time elapsed since entering the face scanning screen 	User scans their face successfully.
<i>Selfie Photo Failure</i>	<ul style="list-style-type: none"> • time elapsed since entering the face scanning screen 	User fails an attempt to scan their face.

ISO 18013-5

Overview

The **Mobile ID SDK** provides the app with all of the functionality needed for ISO 18013-5 compliance as the mDL holder. The compliance is defined as the ability to share personal information in a secure and privacy-conscious manner with an mDL reader.

Configuration

In order to use the ISO 18013-5 sharing functionality, the app needs to obtain and store an appropriate data group from the Issuing Authority during the enrollment process.

To make the app request the data group, provide its identifier in the configuration structure related to ISO 18013-5 as shown:

```
let isoConfiguration = ISO18013Configuration(dataGroup:
"application/vnd.idemia.simple.18013-5+v2")

let configuration = MobileIDConfiguration(
    ...
    iso18013: isoConfiguration
    ...
)

let mobileID = try MobileID(configuration: configuration)
```

The ISO 18013-5 functionality can be accessed via a property of the `MobileID` class:

```
import ISO18013

let iso = mobileID.iso
```

Session

An ISO 18013-5 sharing session is represented by the `Session` class. An instance of that class can be obtained by calling a factory method:

```
let session = try iso.createSession()
```

Engagement

The first step of the sharing session is known as device engagement and is composed of two parts:

- Starting the session, which basically means launching background services responsible for actual data transfer. Currently, Bluetooth sharing is supported.
- Delivering engagement data to the reader. The ways to do that depend on the session and the capabilities of your device. Currently, QR code is supported.

```
switch session.deviceEngagementOptions[0] {
    case .qrCode(let payload):
        // render a QR code with the payload and present it on screen
}

session.start(delegate: self)
```

The device engagement structure contains instruction for the reader on how to connect to the mobile app: what medium is used, what are the service characteristics (in case of BLE) and so on.

Session Delegate

When starting the session you need to appoint a `SessionDelegate` - an object which will be receiving session events. These include:

- session start/end
- receiving a request from the reader
- encountering an error

mDL Reader Requests

The mDL reader will send requests to the mobile app (the mDL holder). A request contains a list of the data elements the reader is asking for and a method for responding, which you can use to prompt the user for a decision.

Regardless of whether the user accepts or declines the release of the data elements, a transfer needs to be made to inform the reader of the decision.

The `respond(decision:)` method of the request returns a `Transfer` object, which has a similar layout to the `Session` class - it needs to be started with a delegate.

The concept of responding to requests is illustrated by the snippet below, which assumes the class has a method named `promptTheUserForDecision(items: Set<DataItemName>)` and conforms to `TransferDelegate`:

```
func session(_ session: Session, hasReceived request: Request) {
    promptTheUserForDecision(items: request.items) { decision in
        let transfer = request.respond(decision: decision)
        transfer.start(delegate: self)
    }
}
```

Transfer Delegate

At this point, after the transfer is initiated, all you need to do is wait for its completion. There are only two methods required by the `TransferDelegate` protocol: one informing you of the progress of the particular transfer, as well as the other to pass on any errors.

MobileID SDK Release Notes

New Features

The new features of Version 0.7.-Alpha1 are:

- Data self-update (client initiated)
 - includes the data update due to app upgrade
 - possible to call by the integrator on demand

Improvements

- `Notifications` module is now part of `MobileID` module
- `IssuanceDataProvider` API has changed, in order to provide convenient access to the data.
- `Enrollment.status` is now asynchronous and is extended with more states (`updateRequired`, `cancelled`).
- Lifecycle events (attributes update, cancellation, reinstate) fetching has been moved to a new method `MobileID.update()`.

Migration Guide

- `MobileID.makeNotifications(:)` has been changed to `MobileID.notifications`. `NotificationsConfiguration` must be passed to `MobileIDConfiguration` instead.
- `IssuanceDataProvider`'s methods changed return types, please refer to `Attributes` and `Media` API.
- `Notifications.fetch(:)` method is no longer performing any actions due to lifecycle events. `MobileID.update()` method must be used instead.

- `Enrollment.status` field has been changed to an asynchronous method in `MobileID`. Mapping of old to new states:
 - `enrolled` -> one of: `active`, `updateRequired`, `cancelled` (please refer to Integration guide and `MobileID.Status` API reference for more details),
 - `unenrolled` -> `notEnrolled`,
 - `enrollmentInProgress` not changed.
 - If a `Enrollment` related type can not be found, add `import Enrollment` on top of the source file.
 - `UpdateFinishedCallback` in `MobileID.events` is now called only when a succesful update occurred. (`Result<UpdateSummary, Error>` replaced with `UpdateSummary`).
 - `MainMenuPlugin*` classes has been moved to `Enrollment` module along with numerous internal types.
-

Deliverables

- [MobileID 0.7.0-ALPHA1 Cocoa Pods spec](#)
 - Binaries:
 - [MobileIDCore 0.7.0-ALPHA1](#)
 - [MobileIDEnrollmentPluginDocumentAuthentication 0.7.0-ALPHA1](#)
 - [MobileIDEnrollmentPluginEmailBinding 0.7.0-ALPHA1](#)
 - [MobileIDEnrollmentPluginEndRegistration 0.7.0-ALPHA1](#)
 - [MobileIDEnrollmentPluginFabricPackage 0.7.0-ALPHA1](#)
 - [MobileIDEnrollmentPluginFaceMatching 0.7.0-ALPHA1](#)
 - [MobileIDEnrollmentPluginFingerCapture 0.7.0-ALPHA1](#)
 - [MobileIDEnrollmentPluginIssuance 0.7.0-ALPHA1](#)
 - [MobileIDEnrollmentPluginPhoneBinding 0.7.0-ALPHA1](#)
 - [MobileIDEnrollmentPluginPrepareLicense 0.7.0-ALPHA1](#)
 - [MobileIDISO18013 0.7.0-ALPHA1](#) (new)
 - [Documentation](#)
 - Integration Guide
 - API Reference
-

Version 0.6.0-Alpha1

New Features

- Fetch identity and retrieval of current enrollment transaction data
 - Notifications of related features, including:
 - Dynamic cancellation
 - Device removal
 - Attributes update
 - Online attribute sharing
 - Online authentication
 - Self-update (after update)
 - Partial force upgrade support (not documented yet)
-

Improvements

- ISO-18013 verification feature documented in Integration Guide
 - Better obfuscation
 - Support for LKMS v1 has been dropped
 - Cocoa pods dependencies has been updated (major update - Alamofire from 4 to 5)
-

Fixes

- MID-11332 SDK should not log on release
 - MID-11411 Cannot continue after type two the same correct emails
 - MID-11456 Initialiser for AdditionalAttributesSubmission is crashing
-

Migration Guide

- `MobileIDConfiguration` has minor changes and its instances or JSON files may need adjustments as shown:
 - in `ipv` object (`businessProcess` removed)
 - in `licenses.lkms` object (`version` type changed from `Int` to `enum`)
 - `MobileID.initialize()` method now has an additional parameter named `completionHandler(Error?)`, and you should postpone interacting with the SDK until this block is called.
 - `MobileID.handleDeepLink(url: URL) -> Bool` has been moved to `Enrollment`, and its signature is changed to `handleDeepLink(_ deepLink: DeepLink) -> Bool`. To transform a `URL` into `DeepLink`, use `DeepLink.init?(url: URL)`
 - `Enrollment.start()` method first parameter has changed from `EnrollmentConfiguration` to `Enrollment.Parameters`
-

Deliverables

- [MobileID 0.6.0-ALPHA1 Cocoa Pods spec](#)
- Binaries:
 - [MobileIDCore 0.6.0-ALPHA1](#)
 - [MobileIDEnrollmentPluginDocumentAuthentication 0.6.0-ALPHA1](#)
 - [MobileIDEnrollmentPluginEmailBinding 0.6.0-ALPHA1](#)
 - [MobileIDEnrollmentPluginEndRegistration 0.6.0-ALPHA1](#)
 - [MobileIDEnrollmentPluginFabricPackage 0.6.0-ALPHA1](#)
 - [MobileIDEnrollmentPluginFaceMatching 0.6.0-ALPHA1](#)
 - [MobileIDEnrollmentPluginFingerCapture 0.6.0-ALPHA1](#)
 - [MobileIDEnrollmentPluginIssuance 0.6.0-ALPHA1](#)
 - [MobileIDEnrollmentPluginPhoneBinding 0.6.0-ALPHA1](#)
 - [MobileIDEnrollmentPluginPrepareLicense 0.6.0-ALPHA1](#)
 - [MobileIDISO18013 0.6.0-ALPHA1](#) (new)
 - [MobileIDNotifications 0.6.0-ALPHA1](#) (new)
- [Documentation](#)
 - Integration Guide
 - API Reference

MobileID SDK Release Notes

Version 0.6.0-Alpha1

This release includes two new features and minor existing API changes.

New Features

- Custom enrollment steps support
 - ISO-18013 verification (undocumented yet)
-

Improvements

- Theme is not global anymore (see Migration Guide).
-

Migration Guide

- `MobileIDSDK` module name has been changed to `MobileID`. Also, the names of the `MobileIDSDK` and `MobileIDSDKCore` podspecs and binaries have been changed.
- `MobileID.init(...)` is not throwing errors or applying themes anymore.
- `MobileID.initialize()` MUST be called after `MobileID.init`. Theme is applied in this method and only to view controllers inside `Theme.MIDNavigationController`.
- An additional parameter `EnrollmentConfiguration` has been added to the `Enrollment.start(...)` method.
- If you need the theme to be applied in enrollment, you MUST pass an instance of `Theme.MIDNavigationController` in the `in rootViewController` parameter in the `Enrollment.start(...)` method.

Deliverables

- [MobileID 0.4.0-ALPHA1 Cocoa Pods spec](#)
 - Binaries:
 - [MobileIDCore 0.4.0-ALPHA1](#)
 - [MobileIDEnrollmentPluginDocumentAuthentication 0.4.0-ALPHA1](#)
 - [MobileIDEnrollmentPluginEmailBinding 0.4.0-ALPHA1](#)
 - [MobileIDEnrollmentPluginEndRegistration 0.4.0-ALPHA1](#)
 - [MobileIDEnrollmentPluginFabricPackage 0.4.0-ALPHA1](#)
 - [MobileIDEnrollmentPluginFaceMatching 0.4.0-ALPHA1](#)
 - [MobileIDEnrollmentPluginFingerCapture 0.4.0-ALPHA1](#)
 - [MobileIDEnrollmentPluginIssuance 0.4.0-ALPHA1](#)
 - [MobileIDEnrollmentPluginPhoneBinding 0.4.0-ALPHA1](#)
 - [MobileIDEnrollmentPluginPrepareLicense 0.4.0-ALPHA1](#)
 - [Documentation](#)
 - Integration Guide
 - API Reference
-

Version 0.2.0-BETA

This is an initial **Mobile ID SDK** release. Its purpose is to evaluate the integration process and try the SDK. It is not intended to be used on production.

New Features

- Enrolling and unenrolling
 - Collecting the enrolled user's attributes and media
-

Known Issues

- Liveness challenge (part of selfie capture) colors are not customizable.
-

Deliverables

- [MobileIDSDK 0.2.0-BETA Cocoa Pods spec](#)
- Binaries:
 - [MobileIDSDKCore 0.2.0-BETA](#)
 - [MobileIDEnrollmentPluginDocumentAuthentication 0.2.0-BETA](#)
 - [MobileIDEnrollmentPluginEmailBinding 0.2.0-BETA](#)
 - [MobileIDEnrollmentPluginEndRegistration 0.2.0-BETA](#)
 - [MobileIDEnrollmentPluginFabricPackage 0.2.0-BETA](#)
 - [MobileIDEnrollmentPluginFaceMatching 0.2.0-BETA](#)
 - [MobileIDEnrollmentPluginIssuance 0.2.0-BETA](#)
 - [MobileIDEnrollmentPluginPhoneBinding 0.2.0-BETA](#)
 - [MobileIDEnrollmentPluginPrepareLicense 0.2.0-BETA](#)
- [Documentation](#)
 - Integration Guide
 - API Reference