Ezio Mobile SDK V4.9

Programmer's Guide for Face ID

May 10, 2019

Contents

Preface

About This Guide

This document explains the features an application needs to provide in order to match the functional and security requirements expected for a mobile banking application based on Ezio Mobile SDK. It gives an overview of the API introduced by the SDK and details on the main features of the SDK with their relevant code snippets for both Java and C interfaces.
This guide also provides simple troubleshooting tips that solve most of the integration problems with the SDK.

Who Should Read this Guide

This guide is intended for Gemalto customers /application developers who are responsible for developing on and integrating with Ezio Mobile SDK.This guide uses the term “application developer” to refer to the people implementing or customizing the product and “user” to refer to the mobile phone end-user of the mobile SDK solution.
This guide includes many references to *Ezio Mobile SDK V4.9 Security Guidelines*. These references are in the form:

- GENxx
  For general security guidelines, platform independent.
- ANDxx
  For Android specific practices.
- IOSxx
  For iOS specific practices.

For More Information

The following table contains the complete list of documents for Ezio Mobile SDK V4.9.

Document	**Description**
Ezio Mobile SDK V4.9 Overview	This serves as an introduction to the Ezio Mobile SDK product. It includes information on the package contents, an overview of the system, and a description of the features and services provided by the SDK.
Ezio Mobile SDK V4.9 Release Notes	This contains detailed release information such as new features, issues fixed, known issues, devices and OS versions tested.
Ezio Mobile SDK V4.9 Migration Guides	This set of documents contains the necessary steps when migrating from an earlier versions to *Ezio Mobile SDK V4.9*.
Ezio Mobile SDK V4.9 Security Guidelines	This contains best and recommended security practices that an application developer should follow.
Ezio Mobile SDK V4.9 API Documentation	This document details the interfaces provided by the Ezio Mobile SDK libraries on Android and iOS platforms

Document	Description
Ezio Mobile SDK V4.9 Programmer's Guide	This guide details the usage of the Ezio Mobile SDK when extending the features, functionalities, and interfaces of Ezio Mobile solution.

You may also refer to the following list of links and documents:

- *Chip Authentication Program - Functional Architecture 2007*(MasterCard)
- Android PRNG Fix
- HOTP RFC
- TOTP RFC
- OCRA RFC
- *Gemalto Dynamic Signature (DS) Specification 1.8*
- *Gemalto Generic SWYS Specification 2.2*
- Android, Invalid Key Exception

Contact Us

For contractual customers, further help is provided in the Gemalto Self Support portal at http://support.gemalto.com or you can contact your Gemalto representative.
Gemalto makes every effort to prevent errors in its documentation. However, if you discover any errors or inaccuracies in this document, please inform your Gemalto representative.

About Ezio Mobile SDK

Ezio Mobile is an enterprise authentication solution for e-banking and e-commerce functions such as one-time password (OTP) management, challenge-responses, transaction data signing, data protection, and out-of-band (OOB) communication. It utilizes the user's mobile device as the security platform. The solution consists of mobile applications based on Ezio Mobile SDK (software development kit).
For the detailed product overview, refer to *Ezio Mobile SDK V4.9 Overview*.

Installing Ezio Mobile SDK

Ezio Mobile SDK must be installed as part of your development platform. The first step is to unzip the contents of the Ezio Mobile SDK release onto the host platform. The steps required to build an application based on the Ezio Mobile SDK are listed under their target platform. To test an application based on Ezio Mobile SDK, various servers are required depending on the feature under test (for example, an authentication server to verify an OTP).
Each platform provides different flavors of Ezio Mobile SDK:

- A debug version with additional debug information to simplify application development.
- A release version where the security configuration is enforced by the version (for example, TLS configuration). As the debugger detection is applied to some features (for example, secure storage), it will halt the execution when a debugger is attached to the application process. In addition, for the Android release version, anti-hooking feature has been included where it crashes the application when hooking is detected. Note that approximately 2% of devices on the field could potentially be detected as being hooked.
- iOS Specific: A debug_nocoverage version which is similar to debug version but without code coverage data. By using this version, application is not forced to enable code coverage.

Ezio Mobile SDK provides the following libraries:

- Android: under `debug/`, and `release/`, different flavors of the following libraries are provided.
  - Java library
    `libidpmobile.jar`
    You can configure the build.gradle file to link the debug and release JAR file correctly: scroll code

    ```
    dependencies {
      debugCompile
    files("${your-ezio-root-dir}/android/debug/libidpmobile.jar")
      releaseCompile
    files("${your-ezio-root-dir}/android/release/libidpmobile.jar")
    }
    ```

  - Native libraries
    Since 4.6, EZIO moves sensitive code from Java to C, a shared library "libidp-shared.so" is included. Also, EZIO will not support mips ABIs. For full flavor of EZIO delivery package, the native libraries include:
    `arm64-v8a/libmedl.so`
    `arm64-v8a/libidp-shared.so`
    `armeabi-v7a/libmedl.so`
    `armeabi-v7a/libidp-shared.so`
    `x86/libmedl.so`
    `x86/libidp-shared.so`
    `x86_64/libmedl.so`
    `x86_64/libidp-shared.so`

- Native libraries for FaceID authentication mode
  arm64-v8a/libfid.so
  arm64-v8a/libopenblas.so
  armeabi-v7a/libfid.so
  armeabi-v7a/libopenblas.so

Note:

Please also refer to `EzioMobileSDK_programmers_guide.pdf` for the usage of JNA library.

- Android: Assets for FaceID authentication mode are provided separately.
  `assets/data/FacesCreateTemplateMediumLite.ndf`
  `assets/data/FacesCreateTemplateSmall.ndf`
  `assets/data/FacesDetectSegmentsFeaturePointsTrack.ndf`
  `assets/data/FacesDetectSegmentsLiveness.ndf`
  `assets/data/FacesDetectSegmentsOrientation.ndf`
- iOS
  - `debug/EzioMobile.framework`
  - `debug_nocoverage/EzioMobile.framework`
  - `release/EzioMobile.framework`

Note:

Resources for FaceID authentication mode:

```
EzioMobile.framework/Resources/FacesCreateTemplateMediumLite.ndf
EzioMobile.framework/Resources/FacesCreateTemplateSmall.ndf
EzioMobile.framework/Resources/FacesDetectSegmentsFeaturePointsTrack.ndf
EzioMobile.framework/Resources/FacesDetectSegmentsLiveness.ndf
EzioMobile.framework/Resources/FacesDetectSegmentsOrientation.ndf
```

All these resource files inside EzioMobile.framework need to be manually copied into the Xcode project and added to the application target in order to make the FaceID authentication mode function well.

On Android Platform

Ezio Mobile SDK library for Android must be included as part of the classpath when building for Android. Android Support Library must be downloaded from Android SDK Manager and imported as a library in the Android project. A typical Android project must have a link or copy of the library in the project's libs directory before issuing a build command from Ant. In Eclipse and Android Studio, a project can link the library by including it as an external JAR file in the project's Java build path. Ezio Mobile SDK for Android requires android.support.v4 library version 23.0 or later so as to support the keypad in Secure Input Service which is `android.support.v4.app.DialogFragment`. The activity which displays the SecurePinPad needs to extend `android.support.v4.app.FragmentActivity`. This library is also required to support Android M for the permission handling.
Native codes are embedded into SDK to enhance the root detection functionality. Besides the JAR files, dynamically linked library files (with extension '.so') are included in the delivery. These files are to be copied or linked to the project. For Eclipse user, copy all the folders (named by its architecture) to the project's libs directory. For Android Studio users, copy them to the `src/main/jniLibs/ directory`.

Note:

Application developer using Android Studio version 2.0 or later, needs to disable the following settings:

- For Mac:
  Android Studio > **Preferences** > **Build/Execution/Deployment** > **Instant Run** > **Enable Instant Run to hot swap code/resource on deploy (default enabled)**
- For Windows:
  File > **Settings** > **Build, Execution, Deployment** > **Instant Run** > **Enable Instant Run to hot swap code/resource on deploy (default enabled)**

Otherwise, the application will crash due to anti-hooking detection in Ezio Mobile SDK V4.8. Alternatively, for development purposes, you can use the debug version of Ezio Mobile SDK which does not perform the hook checks.

Warning:

- Ezio Mobile SDK uses some immutable characteristics of the platform and the application's package name to initialize its environment. By default, the application that uses Ezio Mobile SDK is sealed to the device on which the Ezio Mobile SDK services are used the first time.
- Ezio Mobile SDK V4.8 ceased support for Android 2.x. It implies that Gemalto does not validate the SDK on Android 2.x and no longer gurantees the functionalities of the SDK on Android 2.x and earlier versions. You have to perform your own validation if it is needed. In addition, obfuscation with Dexguard has some issues on Android 2.3 when it tries to write mapped classes/data in the SDCard. In order to make the application work for earlier version, the following codes should be added before initializing Ezio Mobile SDK.

```
System.setProperty("java.io.tmpdir",
                    getDir("files",
                    Context.MODE_PRIVATE).getPath());
```

- The latest version of Dexguard has been used to obfuscate Ezio Mobile SDK V4.6. Some devices may encounter issues when Dexguard tries to write temporary files onto the device's SDCard. As such, it is recommended that application makers do one of the following:
  a. Grant the WRITE_EXTERNAL_STORAGE and READ_EXTERNAL_STORAGE in the Android manifest file as follows:
  ```
  <uses-permission
  android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
  <uses-permission
  android:name="android.permission.READ_EXTERNAL_STORAGE"/>
  ```
  b. `Include the following codes in your application before initializing Ezio Mobile SDK. System.setProperty("java.io.tmpdir", getDir("files", Context.MODE_PRIVATE).getPath());`
  c. If your application using proguard/dexguard, refer to `proguard-project.pro` inside EZIO Mobile SDK delivery package for the proper configuration.

On iOS Platform

The following frameworks have to be added to your project in order to link your application with Ezio Mobile SDK:

- `Foundation.framework`
- `SystemConfiguration.framework`
- `UIKit.framework`
- `LocalAuthentication.framework`
- `Security.framework`
- `AVFoundation.framework`
- `CoreMedia.framework`
- `Accelerate.framework`

Also, add the following dynamic library to your project:

- `libc++.dylib`
- `libsqlite3.0.dylib`

Add the following property into the `info.plist` file.

- Privacy - Camera Usage Description

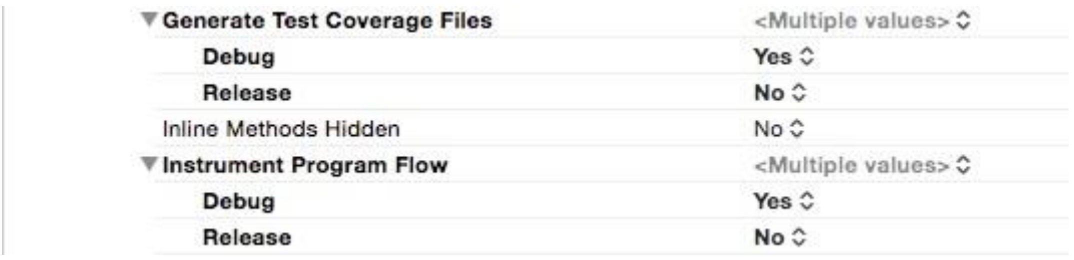
The compilation option `-ObjC` must also be added in the **other linker flags** item under the Xcode project build settings including `-ObjC -all_load`. In addition, the following security guidelines \[SG\] must be enforced for security when setting the application project:

- For IOS06, remove the symbols from Xcode output.
- For IOS07, enable Xcode compiler security and obfuscation options.

To be able to use the debug version of Ezio Mobile framework and prevent errors during the build, you may want to use the `debug_nocoverage` framework or set two options in the **Build Settings** section of your project:

- Generate Test Coverage Files: "YES" for the **Debug** version, "NO" for the **Release** version.
- Instruments Program Flow: "YES" for the **Debug** version, "NO" for the **Release** version.

Figure 1 Option for Build Settings



▼ **Generate Test Coverage Files**	<Multiple values>
Debug	**Yes**
Release	**No**
Inline Methods Hidden	No
▼ **Instrument Program Flow**	<Multiple values>
Debug	**Yes**
Release	**No**

FaceID Authentication

This section describes using the FaceID authentication built in Ezio Mobile SDK to perform the enrollment and authentication using the user's face.
A sample application is included in the Ezio Mobile SDK delivery package. It works as a reference for developer to handle FaceID user interface.

Note:

This feature is only available in Ezio Mobile SDK Full release package. Please contact Gemalto sales representatives to enable this feature.

Application Requirements

Supported Devices and Operating Systems

The face authentication feature is supported on the following devices with the operating systems (OS):

- iOS V8.0 and later.

  Note:

  Starting from Ezio Mobile SDK V4.6, the Gemalto Face ID only supported on the device that does not have System Face ID to comply with Apple App Store new rules.

- Android V4.4 and later, with "arm" processor.

In addition, the mobile device must have a camera and preferably a front camera which can be easily used to capture the user's face features while using the application.

Android Permission

On Android, some permissions are required to be set before you can use this feature in your application. Ensure that the following permissions are added in the Android manifest file.

```
<uses-permission android:name="android.permission.CAMERA"/>
<uses-feature android:name="android.hardware.camera"/>
```

Note:

The application using Android M onwards will request for the permission at runtime. It is recommended to set target SDK level to at least 23, and always check and perform the permission requests at the launch of the application.

Requirement Checks

The following checks are to be performed before Ezio Mobile SDK can provide the FaceID authentication.

Checking if device supports FaceID

The device supportability is checked. Unsupported devices/OS version will cause your app to crash if it attempts to use the feature.
On Android:

```
boolean faceAuthSupported = faceAuthService.isSupported();
```

On iOS:

```
BOOL faceAuthSupported = [faceAuthService isSupported:&error];
```

Checking if the FaceID license is configured

User has to obtain a valid FaceID license before using the FaceID authentication mode feature.
On Android:

```
boolean licenseConfigured = faceAuthService.isLicenseConfigured();
```

On iOS:

```
BOOL licenseConfigured = [faceAuthService isLicenseConfigured];
```

Checking if the service is initialized

The FaceID authentication has to be initialized first in order for the camera to be accessed by the application.
On Android:

```
boolean faceAuthInitialized = faceAuthService.isInitialized();
```

On iOS:

```
BOOL faceAuthInitialized = [faceAuthService isInitialized];
```

Checking if user's face is configured

User must have performed the face enrollment on the device.
On Android:

```
boolean faceAuthConfigured = faceAuthService.isConfigured();
```

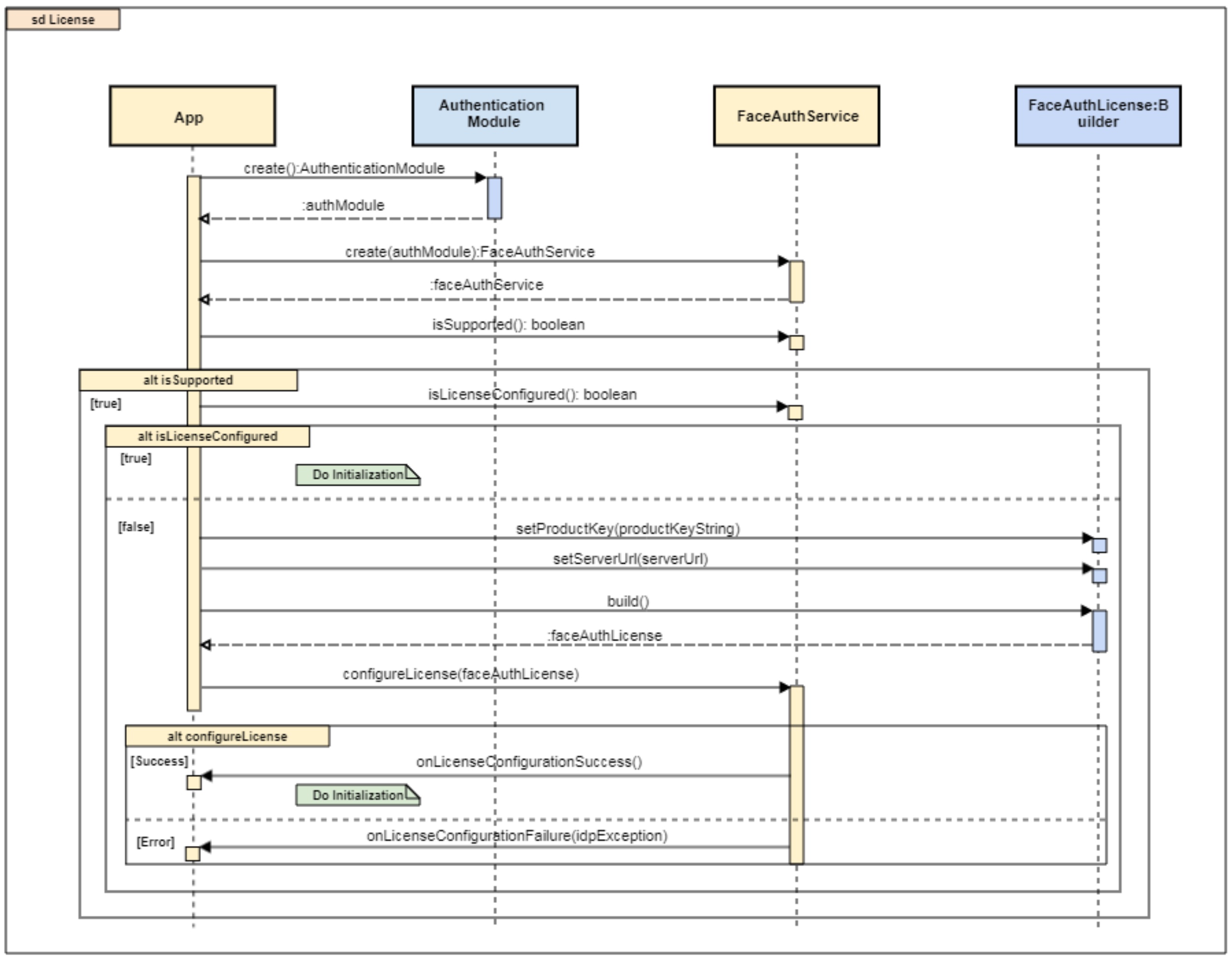
On iOS:

```
BOOL faceAuthConfigured = [faceAuthService isConfigured:&error];
```

Service Creation and License Configuration

In order to access the functions of the FaceID authentication, the `FaceAuthService` object has to be created first and the license has to be configured before calling the initialize function from `FaceAuthService`.

Figure 2 Service Creation and License Configuration



On Android:

```
// Create AuthenticationModule.
// It's the entry point for all authentication related features.
```

```
AuthenticationModule authModule = AuthenticationModule.create();

// Create an object that represents face authentication service FaceAuthService
faceAuthService = FaceAuthService.create(authModule);


if(faceAuthService.isSupported()){
    if(!faceAuthService.isLicenseConfigured()) {
         // Configure the license
         final FaceAuthLicense.Builder builder =
                                      new FaceAuthLicense.Builder();
         FaceAuthLicense license = builder
                         .setProductKey("productKeyString")
                         .setServerUrl(https://serverurl.com)
                         .build();

        faceAuthService.configureLicense(license,
              new FaceAuthLicenseConfigurationCallback() {
            @Override
             public void onLicenseConfigurationSuccess() {
                 // do initialization             }
            @Override
             public void onLicenseConfigurationFailure
                 (IdpAuthException e) {
                 // on Failure, error handling
             }
         });
     } else {
         // do initialization
     }
} else {
     // feature is not supported on this device
}
```

On iOS:

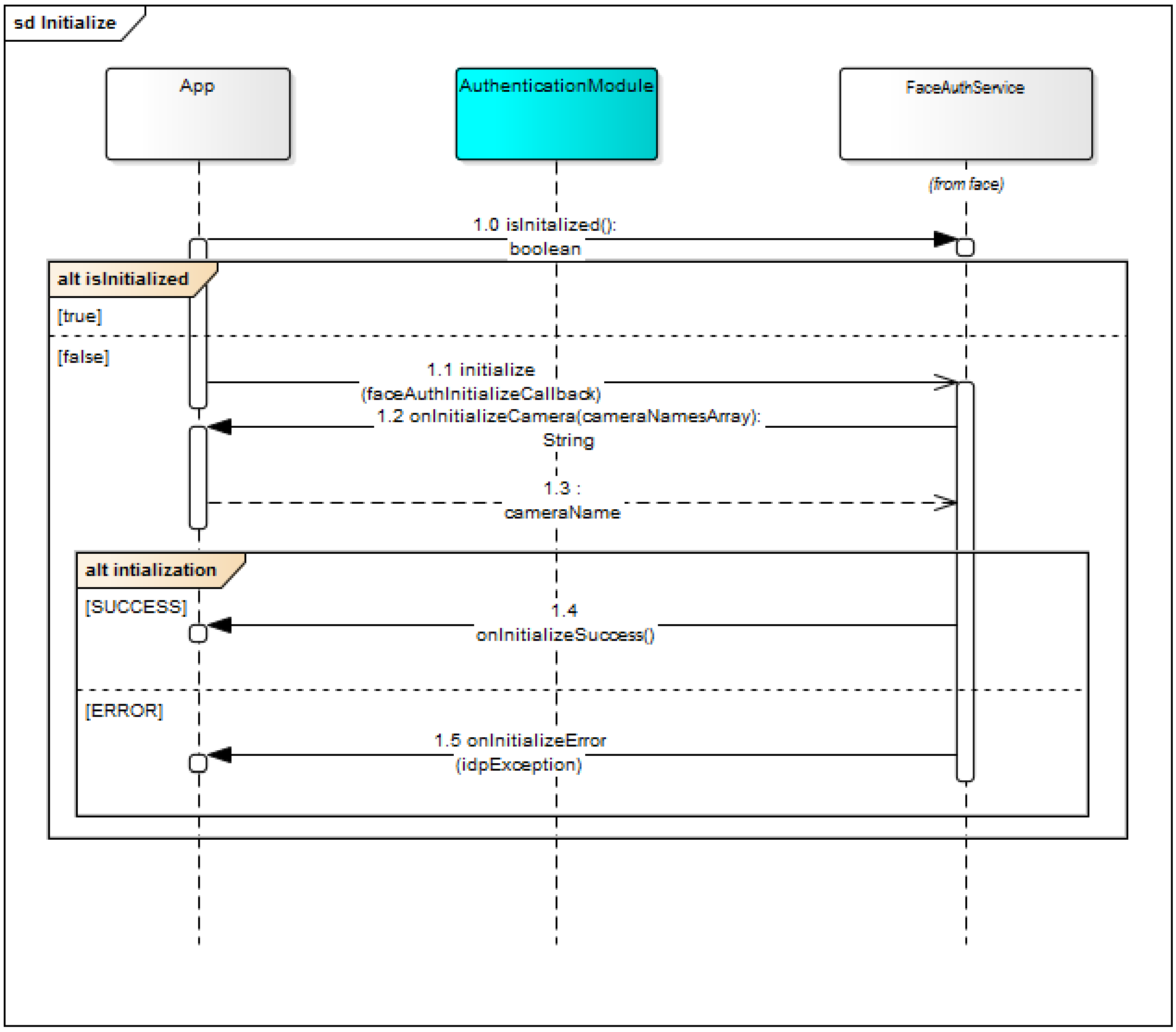
```
// Create AuthenticationModule.
// It’s the entry point for all authentication related features.
EMAuthModule *authModule = [EMAuthModule authModule];
// Create an object that represents face authentication service
EMFaceAuthService *faceAuthService =
                [EMFaceAuthService serviceWithModule:authModule];


NSError *error;

if ([faceAuthService isSupported:&error]) {
     if(![faceAuthService isLicenseConfigured]) {
         // configure the license
         [faceAuthService configureLicense:
             ^(EMFaceAuthLicenseBuilder *builder) {
             builder.productKey = @productKeyString”;
             builder.serverUrl = @https://serverurl.com;
```

```
        } completion:^(BOOL success, NSError *error) {
            // do initialization if success is true
         }];      } else {
         // do initialization
     } } else {
     // feature is not supported on this device
 }
```

Initializing FaceAuthService

After the service has been created and the license has been configured, it is required to perform an initialization to check the cameras for face template enrollment or face verification. This process may take a few seconds, depending on the device used.

Figure 3 Service Initialization



sd Initialize
App
AuthenticationModule
FaceAuthService
(from face)
1.0 isInitalized(): boolean
alt isInitialized
[true]
[false]
1.1 initialize (faceAuthInitializeCallback)
1.2 onInitializeCamera(cameraNamesArray): String
1.3 : cameraName
alt intialization
[SUCCESS]
1.4 onInitializeSuccess()
[ERROR]
1.5 onInitializeError (idpException)

On Android:

```
if(faceAuthService.isSupported && !faceAuthService.isInitialized()){
     // initialize the library
     faceAuthService.initialize(new FaceAuthInitializeCallback() {

        @Override
         public void onInitializeSuccess() {
             // on success, you can now perform enrollment
              // and verification feature
         }
        @Override
         public void onInitializeError(IdpException arg0) {
             // on Fail, error handling
              // (if can be caused by eg. missing permission)          }
        @Override
         public String onInitializeCamera(String[] arg0) {
             // Select one from the given list by returning null,
              // the SDK will pick a default camera which will be:
             // the first in the list which contains 'front'
             // or the first one if no 'front' is found
             return null;
         }
     });
 }
```

On iOS:

```
BOOL faceIdSupported = [faceAuthService isSupported:&error];
BOOL faceIdInitialized = [faceAuthService isInitialized];

// Init face id service
 if (faceIdSupported && !faceIdInitialized) {
     [faceAuthService initializeWithCompletion:
          ^(BOOL isInitialized, NSError *error) {
             if (error) {
                 // Handle error
             }
             // Check isInitialized status
     }];
}
```

Un-initializing FaceAuthService

It is recommended to perform an uninitialization once the user has finished using the feature in order to release the resource used.
On Android:

```
if(faceAuthService.isInitialized()){
     // uninitialize the library
     faceAuthService.uninitialize();
}
```

On iOS:

```
if ([faceAuthService isInitialized]) {
     [faceAuthService
          uninitializeWithCompletion:^(BOOL success, NSError *error) {
             if (error) {
                 // Handle error
             }
             // Check success
     }];
}
```

Upgrading to Multi-Authentication Mode

By default, the tokens are authenticated via the PIN mode. If the device supports the FaceID authentication, the token can be upgraded to support the multi-authentication mode using the token's PIN, before FaceID authentication is activated and used.

Note:

The correct PIN has to be presented when the token is upgraded to support multi-authentication mode. An OTP verification is recommended before the token is upgraded.

Activation and Deactivation

Activation Check

The following snippets are used to check if the FaceID authentication mode is activated.
On Android:

```
// Get object that represents face authentication functionality
FaceAuthMode faceAuthMode = faceAuthService.getAuthMode();
boolean bIsAuthModeActive = token.isAuthModeActive(faceAuthMode);
```

On iOS:

```
// Get object that represents face authentication functionality
BOOL bIsAuthModeActive = [token isAuthModeActive:[faceAuthService authMode]];
```

Activation

Once the multi-authentication mode is enabled on the token, the FaceID authentication mode can be activated via the FaceAuthMode class. The following code snippets demonstrate the activation of the FaceID authentication mode.
On Android:

```
// Get object that represents face authentication functionality FaceAuthMode
faceAuthMode = faceAuthService.getAuthMode();
 if (!token.isAuthModeActive(faceAuthMode)) {
     // Activate face authentication mode
     // The token's pin must be passed
     // in order to activate face authentication mode
     token.activateAuthMode(faceAuthMode, pinAuthInput);
}
```

On iOS:

```
 // Get object that represents face authentication functionality
// Check the mode is activated or not
if (![token isAuthModeActive:\[faceAuthService authMode]]){
    // Activate face authentication mode
    // The token's pin must be passed
    // in order to activate face mode
    BOOL result = [token activateAuthMode:[faceAuthService authMode]
                  usingActivatedInput:pinAuthInput
                                error:&error];
    if (!result) {
        // Handle error
    }
}
```

Deactivation

The following code snippets demonstrate the deactivation of the FaceID authentication mode.
On Android:

```
// check if token is in multi auth mode
// and if faceId mode is active if (token.isMultiAuthModeEnabled()
      && token.isAuthModeActive(faceAuthMode)) {
     token.deactivateAuthMode(faceAuthMode);
}
```

On iOS:

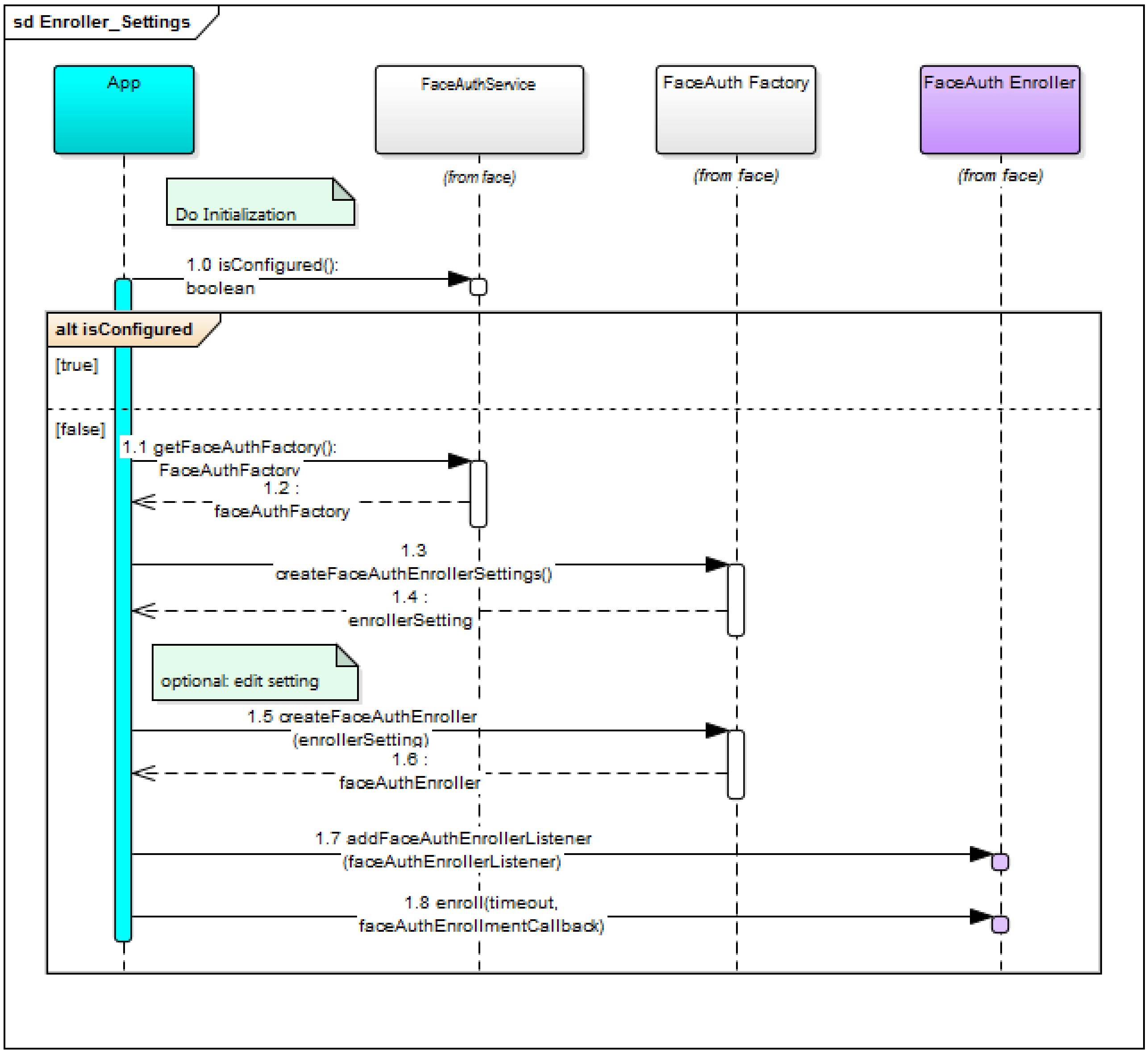
```
// check if token is in multi auth mode
// and if faceId mode is active
if ([token isMultiAuthModeEnabled] &&
    [token isAuthModeActive:\[faceAuthService authMode]]) {
    BOOL result=[token deactivateAuthMode:[faceAuthService authMode]
                                error:&error];
    if (!result) {
        // Handle error
   }
}
```

FaceID Enrollment

Creation of FaceAuthEnroller

`FaceAuthEnroller` is the interface for performing FaceID enrollment related activities. Currently, Ezio Mobile SDK supports only one face template. The following figure demonstrates the checks before setting up the enroller.

Figure 4 Creation of Enroller


sd Enroller_Settings
App
FaceAuthService
(from face)
FaceAuth Factory
(from face)
FaceAuth Enroller
(from face)
Do Initialization
1.0 isConfigured(): boolean
alt isConfigured
[true]
[false]
1.1 getFaceAuthFactory(): FaceAuthFactory
1.2 : faceAuthFactory
1.3 createFaceAuthEnrollerSettings()
1.4 : enrollerSetting
optional: edit setting
1.5 createFaceAuthEnroller (enrollerSetting)
1.6 : faceAuthEnroller
1.7 addFaceAuthEnrollerListener (faceAuthEnrollerListener)
1.8 enroll(timeout, faceAuthEnrollmentCallback)

On Android:

```
FaceAuthFactory faceAuthFactory = faceAuthService.getFaceAuthFactory();
 FaceAuthEnrollerSettings enrollerSettings =
         faceAuthFactory.createFaceAuthEnrollerSettings();


// Option 1: Create enroller with setting FaceAuthEnroller enroller =
         faceAuthFactory.createFaceAuthEnroller(enrollerSettings);
```

```
// Option 2: Create default enroller FaceAuthEnroller defaultEnroller =
         faceAuthFactory.createFaceAuthEnroller();
```

On iOS:

```
// Create face auth factory
EMFaceAuthFactory* faceAuthFactory = [faceAuthService faceAuthFactory];

// Option 1: Create Enroller with setting
EMFaceAuthEnrollerSettings *enrollerSetting =
       [faceAuthFactory createFaceAuthEnrollerSettings];
// Set the seconds before the image capture starts.
enrollerSetting.coundownToCapture = 3;
// Set the number of frames to enroll.
enrollerSetting.numberOfFramesToEnroll = 3;

id<EMFaceAuthEnroller> enroller =
 [faceAuthFactory createFaceAuthEnrollerWithSettings:enrollerSetting];

// Option 2: Create default enroller
id<EMFaceAuthEnroller> defaultEnroller =
       [faceAuthFactory createFaceAuthEnrollerWithDefaultSettings];
```

Setting up Listener and Callbacks

Once the enroller is created, the listener and callback have to be created. Listener is used to handle frame event to determine the image/frame to be registered for the enrollment.
On Android:

```
FaceAuthEnrollerListener lListener =
    new FaceAuthEnrollerListener() {

    @Override
    public void onFrameReceived(FaceAuthFrameEvent event) {
        // Handle frame event
    }
}
// Add the listener
enroller.addFaceAuthEnrollerListener(listener);

// Start to enroll the user with face.
// The result will be returned from the callbacks.
int timeout = 30000; // timeout in millisecond
enroller.enroll(timeout, new FaceAuthEnrollerCallback() {
    @Override
    public void onEnrollFinish(FaceAuthStatus status) {
        // enrollment done.
        // the status indicates the enrollment result.
        // Note that onEnrollFinish does not
        // guarantee the success of enrollment
        enroller.removeFaceAuthEnrollerListener(listener);
    }
    @Override
    public void onEnrollError(IdpException e) {
        // handle error
        enroller.removeFaceAuthEnrollerListener(listener);
    }
});
```

On iOS:

```
// Set delegate method
[enroller setFaceAuthEnrollerDelegate:enrollerDelegate];


// Start the enrollment process. The completion block
// will be running on main thread.
[enroller enroll:30000 withCompletion:
     ^(BOOL success,id<EMFaceAuthEnroller> enroller,NSError *error){
         if (error) {
             // handle the error case
         } else {
         }
        if (!success) {
             // Update UI components, e.g. prompting an alter.
         }
             // Remove delegate method
            [enroller removeFaceAuthEnrollerDelegate];
}];
```

Handling of the FrameEvent

The current enrollment process is manual and the application has to determine if a frame is to be added, and if there are enough frames to complete the enrollment. Ezio Mobile SDK will call back to the application with a frame event whenever a frame has been captured by the SDK.

Visual Feedback

The frame event contains the image that can be directly displayed to the user to provide a visual feedback based on what has been captured by the camera.
The following code snippets demonstrate how to display the feedback to the user.
On Android:

```
@Override
public void onFrameReceived(FaceAuthFrameEvent event) {
  // always display the image to the user
  // to provide visual feedback
  // FaceView class is provided by the SDK.
  // It needs to be declared in your layout.xml  faceView.setFaceFrameEvent(event);
 //Alternatively, you can use the default Android ImageView widget
imageView.setImageBitmap(event.getImage().toBitmap());
 ...
}
```

On iOS:

```
// required function from EMFaceAuthEnrollerDelegate protocal
// Handle enrollment process here
- (void)enroller:(id<EMFaceAuthEnroller>)enroller
didUpdateFaceAuthFrameEvent:(id<EMFaceAuthFrameEvent>)frameEvent
{
     // Get the captured image to display on UIImageView.
     UIImage *image = [frameEvent image];
}
```

On iOS, Ezio Mobile SDK also provides count down delegate methods to facilitate user handle UI effects.

```
// Notifies the user of the countdown in seconds
// before the actual capturing is performed
- (void)enroller:(id<EMFaceAuthEnroller>)enroller
didCountdownToCaptureWithSeconds:(NSTimeInterval)timeInterval
{
     // Add some UI handler, e.g. provides a visual effect
     // of counting down before the real capture starts.
}
// The actual capturing of face has started
- (void)enrollerDidStartCapturing:(id<EMFaceAuthEnroller>)enroller
{
     // Add some UI handler, e.g. Informing user that
     // the app is doing face capture.
}
// The capturing has ended
- (void)enrollerDidEndCapturing:(id<EMFaceAuthEnroller>)enroller
{
     // Add some UI handler, e.g. Informing user that
     // the app finished face capture.
}
```

Response to EventFrame Information

The frame event also contains other information to allow the application to decide if the frame is in good form to be extracted as part of the template.

- Face coordinates
  The coordinates of the face with respect to the complete face image. It gives information about the size of the actual face as well as the position of the face. The application checks if the actual face is centered within the rectangle coordinates and if the size of the face is satisfiying. Users may be prompted to center their faces or to move closer to the camera if needed.
- Face yaw/pitch/roll angle
  The angles detected on the face rotates along 3 axes. For example, a value "0" indicates that the front face is captured. If the application wants to capture the user face from the left/right side, it will prompt the user to turn left/right and the yaw angle is checked against the predefined threshold angle.
  On Android:

```
@Override
public void onFrameReceived(FaceAuthFrameEvent event) {
  // Application is advised to check the following
  // before the frame is added:
  // (Recommended) check face coordinates Rect faceCoordinate =
event.getFaceCoordinate();
  if(faceCoordinate.bottom == 0 && faceCoordinate.top == 0  &&
faceCoordinate.left == 0 && faceCoordinate.right == 0){
  // face not detected, prompt the user
  return;
  }
  //(Optional) checks on the face orientation
  //to see if the face it well oriented.
  //eg. pitch/roll/yaw close to 0 implies that it is a front face


   float pitch = event.getPitchAngle();
   float roll = event.getRollAngle();
   float yaw = event.getYawAngle();

  // Once all checks are performed and application
  // decides it is a good template
  // add it to the SDK for extraction
  try {
      enroller.addFaceFrameForEnrollment(event);
  } catch (FaceAuthException e) {
      // handle error
  }  ...
 }
```

On iOS:

```
- (BOOL)enroller:(id<EMFaceAuthEnroller>)enroller
shouldAddFaceEvent:(id<EMFaceAuthFrameEvent>)frameEvent
          withIndex:(NSUInteger)index
{
     BOOL addFaceEvent = NO;
    // Examine frame quality from each frame event
    // and return YES if quality is good. The quality
    // benchmark is defined by application.
   // Application is advised to check the following
   // attributes before the frame is added,
   // (Recommended) check face coordinates.
    CGRect imageBound = [frameEvent imageBounds];
     if(imageBound.size.height==0 && imageBound.size.width==0){
         // Face not detected.
         return NO;
     }
    // (Optional) Check on the face orientation to see
    // if the face is well oriented.
    // E.g. When the angles for pitch, roll and yaw are
    // close to ZERO, it means a front face is detected.
```

```
        float yaw = [frameEvent yawAngle];
        float pitch = [frameEvent pitchAngle];
        float roll = [frameEvent rollAngle];
              return addFaceEvent;
     }
```

- Face extraction status

  A frame that has been previously marked for extraction. The next frame event will contain the information of the extraction status—fail, still extracting or success. The application keeps track of the successful extractions and invoke a complete enrollment call once the expected number of face images is reached.

  On Android:

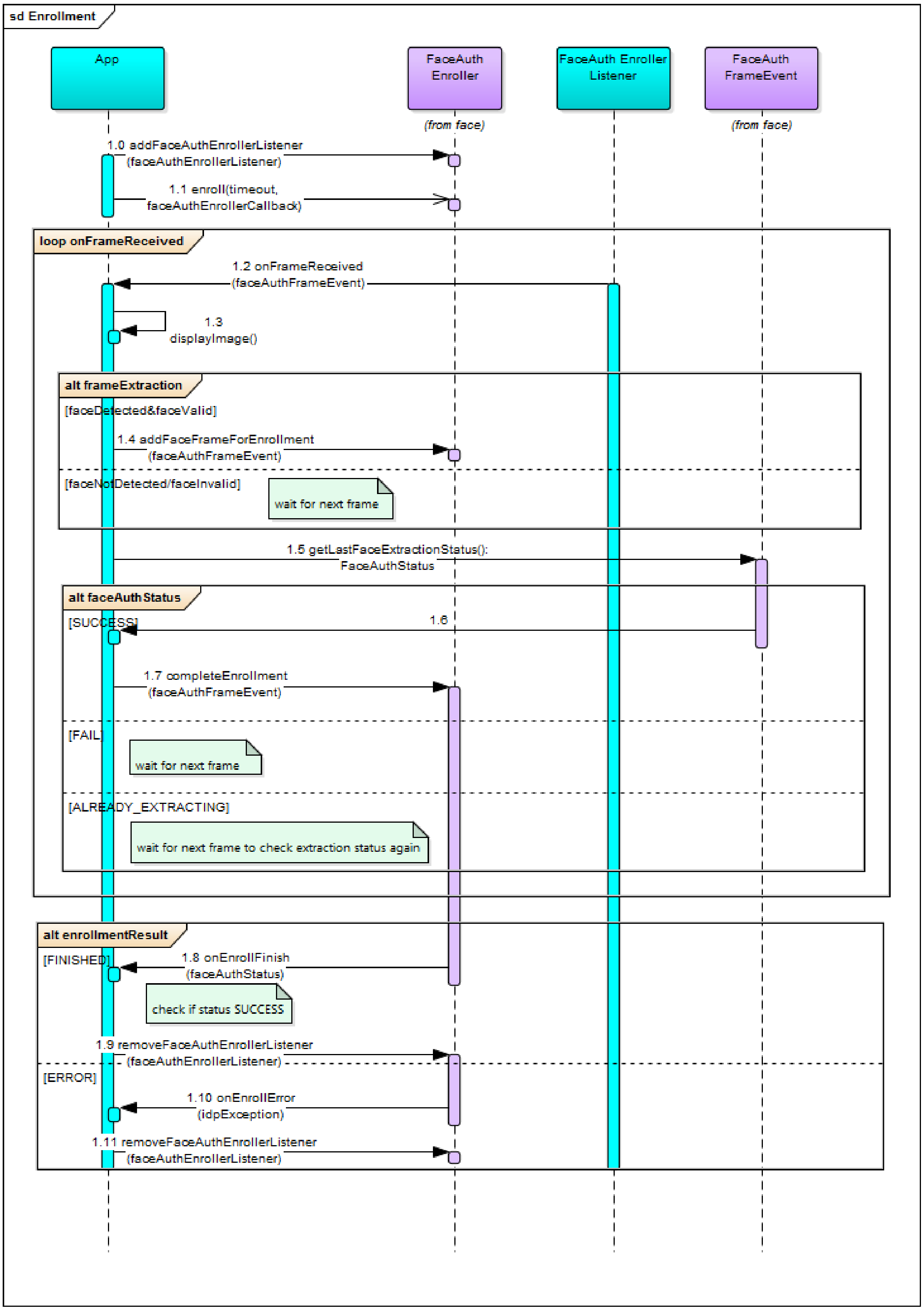
```
    @Override
    public void onFrameReceived(FaceAuthFrameEvent event){
    ...
    FaceAuthStatus status = event.getLastFaceExtractionStatus();

        if(status ==  FaceAuthStatus.SUCCESS){
             // if application maintains the number of images
             // to be added, increment it.
             // if enough images as been extracted
             // (number is decided by application: at least 1)
             // complete the enrollment, it will go to
             // onEnrollFinish(FaceAuthStatus) call
             enroller.completeEnrollment(event);
         } else if (status == FaceAuthStatus.ALREADY_EXTRACTING){
             // Still extracting so wait for next
             // frame event to do the same check
         } else {
             // Failed, check status for different error
         }
        ...
    }
```

  On iOS:

  The extraction status is internally handled on the iOS platform. All the checks mentioned are handled by the application. If a face is not detected in the image, it will still be marked for extraction, and it will lead to a verification failure later on.

  The complete enrollment flow is illustrated in the following figure.

Figure 5 Enrollment Flow Sequence


sd Enrollment
App
FaceAuth Enroller
(from face)
FaceAuth Enroller Listener
FaceAuth FrameEvent
(from face)
1.0 addFaceAuthEnrollerListener (faceAuthEnrollerListener)
1.1 enroll(timeout, faceAuthEnrollerCallback)
loop onFrameReceived
1.2 onFrameReceived (faceAuthFrameEvent)
1.3 displayImage()
alt frameExtraction
[faceDetected&faceValid]
1.4 addFaceFrameForEnrollment (faceAuthFrameEvent)
[faceNotDetected/faceInvalid]
wait for next frame
1.5 getLastFaceExtractionStatus(): FaceAuthStatus
alt faceAuthStatus
[SUCCESS]
1.6
1.7 completeEnrollment (faceAuthFrameEvent)
[FAIL]
wait for next frame
[ALREADY_EXTRACTING]
wait for next frame to check extraction status again
alt enrollmentResult
[FINISHED]
1.8 onEnrollFinish (faceAuthStatus)
check if status SUCCESS
1.9 removeFaceAuthEnrollerListener (faceAuthEnrollerListener)
[ERROR]
1.10 onEnrollError (idpException)
1.11 removeFaceAuthEnrollerListener (faceAuthEnrollerListener)

Enrollment Cancellation and Cleanup

The enrollment process is cancelled if the user decides to terminate the process while it is still in progress. With the cancellation, the frame event listener must be removed by calling the corresponding function.
On Android:

```
// any time when user cancels during the enrollment,
// the following needs to be called
// and the listener MUST be removed
enroller.cancelEnrollmentProcess();
enroller.removeFaceAuthEnrollerListener(faceAuthEnrollerListener);
```

On iOS:

```
// any time when user cancels during the enrollment,
// the following needs to be called
// and the delegate MUST be removed

//cancel enrollment
[enroller cancel];

// Remove delegate method
[enroller removeFaceAuthEnrollerDelegate]
```

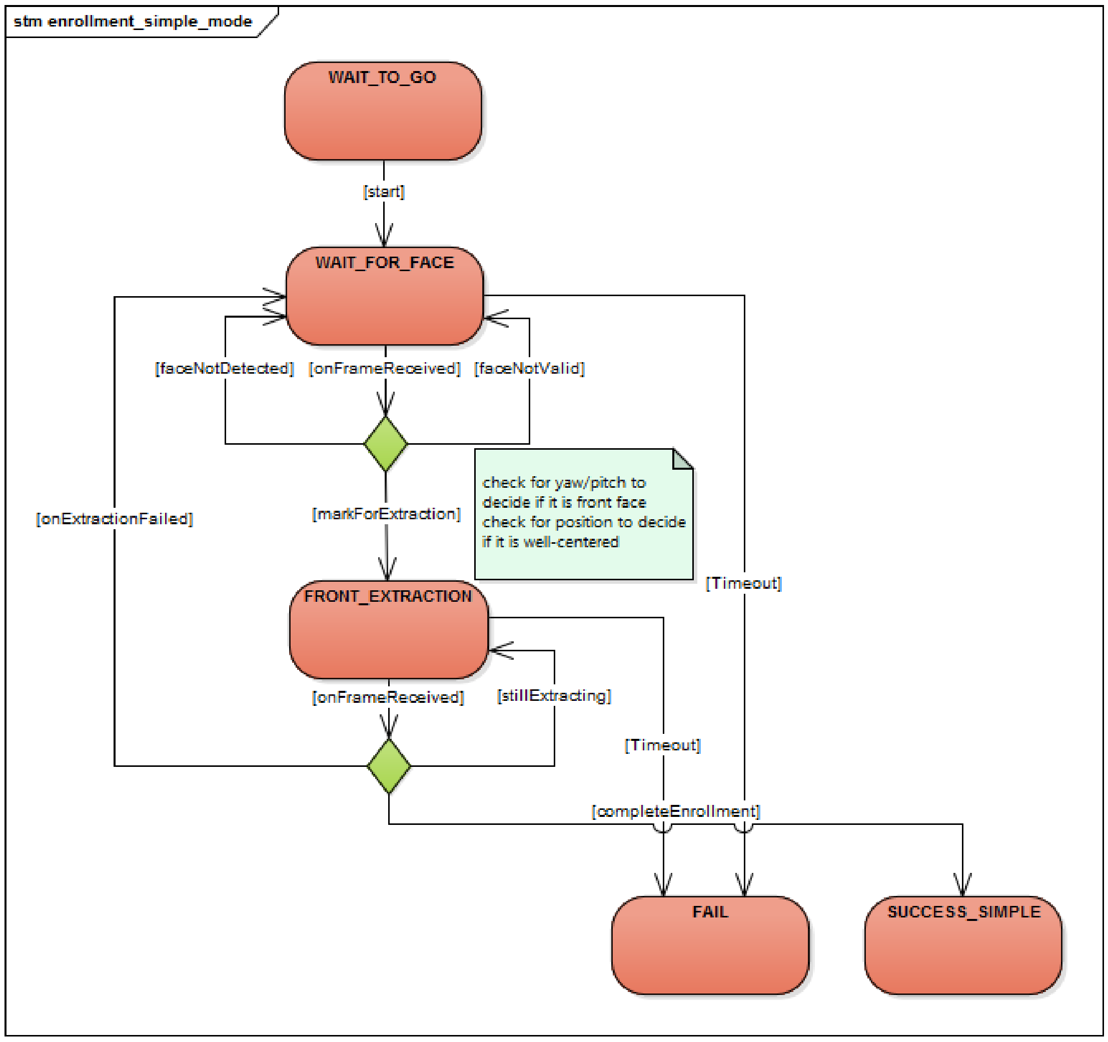
Note:

Failure to remove the listener may cause potential memory leaks and malfunctioning of the FaceID authentication feature.

Enrollment Process

The following example demonstrates the two modes used in the enrollment—simple mode and full mode.

Figure 6 Simple Mode Enrollment



1. Once the enrollment is called, Ezio Mobile SDK starts capturing and passing the application with frame events. On each frame event, application should display the image to provide visual feedback to the user.
2. Once a frame event is received, the application checks for the face coordinates to decide if a human face is detected. The coordinates contains the four corners—left, right, top, bottom of the bounding rectangle. If all the coordinate values are "0", this implies that a face is not detected. The application waits for the next frame event while prompting the user to readjust the position of the face to be within the camera.
3. Once a face is detected, the application checks the coordinates of the face image again to decide if it is well-centered and well-sized. If the face is in good position, proper size and correct face orientation, the application will ask the enroller to extract the current face image.
4. Once a face is marked for extraction, the application waits for the next frame by checking the status of the extraction. Repeat the previous step if the extraction fails.

   Note:

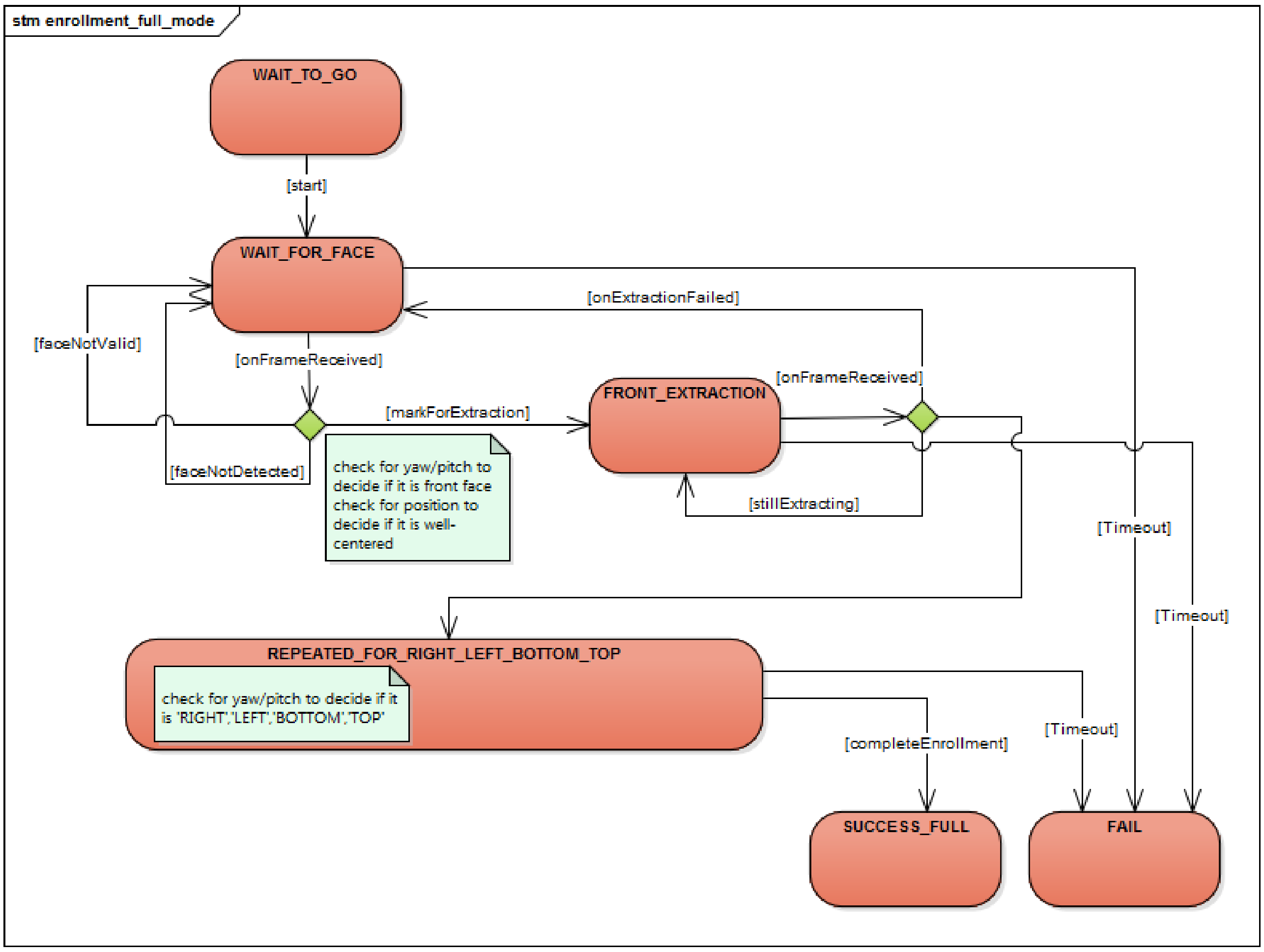
   Do not move if the face is still in the process of extraction.

   The application will call the enroller to complete the enrollment when the face is extracted successfully.
5. In the earlier step, only one front image is extracted for the face template. The application can also choose to have more images for the template by keeping count of the extracted images. The full mode enrollment

from the example demonstrates how an application can extract face images captured from front, left, right, bottom and top views.

The full mode enrollment is illustrated in the following figure.

Figure 7 Full Mode Enrollment



The steps in full mode enrollment is similar to that in the simple mode when the front face image is extracted. However, the application does not call the enrollment completion but continues to process the frame events. For the next face detected, it checks for the yaw angle to ensure that it is captured from the right side of the user face. Until it is successfully extracted, then the application request to capture the left side, followed by bottom and top. Each step of the extraction is the same as the simple mode but it checks for a different yaw/pitch angle in order to be extracted.

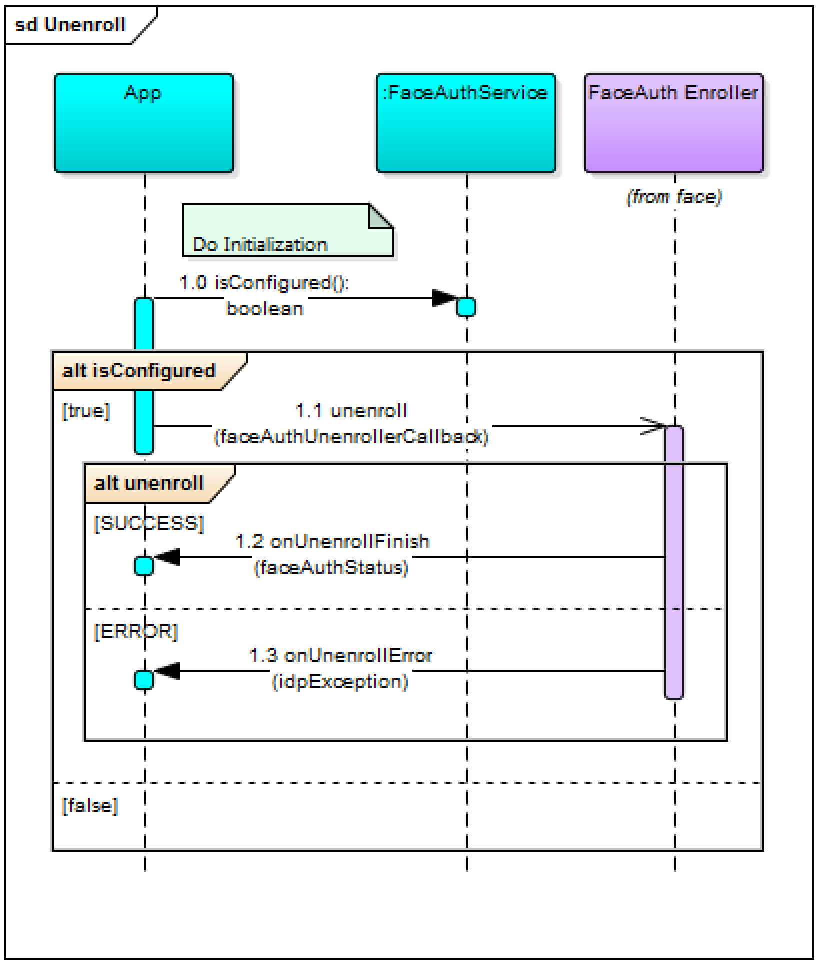
Note:

On iOS, Ezio Mobile SDK internally handles the number of frames to be captured, which has a default value of **4**. User can overwirte this value by setting property `numberOfFramesToEnroll` of object `EMFaceAuthEnrollerSettings`. User also can set the seconds counting down before the capture starts by setting property `countdownToCapture`.

Unenrollment

Unenrollment is used to remove the facial data that has been registered to the device previously. The FaceID authentication mode that has been previously activated will be deactivated for all the tokens on device.

Figure 8 Flow of Unenrollment



sd Unenroll
App
:FaceAuthService
FaceAuth Enroller
(from face)
Do Initialization
1.0 isConfigured(): boolean
alt isConfigured
[true]
1.1 unenroll (faceAuthUnenrollerCallback)
alt unenroll
[SUCCESS]
1.2 onUnenrollFinish (faceAuthStatus)
[ERROR]
1.3 onUnenrollError (idpException)
[false]

The following code snippets demonstrate how to perform the unenrollment process.
On Android:

```
// check if there is enrollment data
if (faceAuthService.isConfigured()) {
     enroller.unenroll(new FaceAuthUnenrollerCallback() {
        @Override
         public void onUnenrollFinish(FaceAuthStatus faceAuthStatus) {
             // unenrollment done         }
        @Override
         public void onUnenrollError(IdpException e) {
             // handle error
         }
     });
 }
```

On iOS:

```
// check if there is enrollment data
if ([faceAuthService isConfigured:&error]) {
     [enroller unenrollWithCompletion:^(BOOL success,NSError *error){
         if (error) {
             // handle error
```

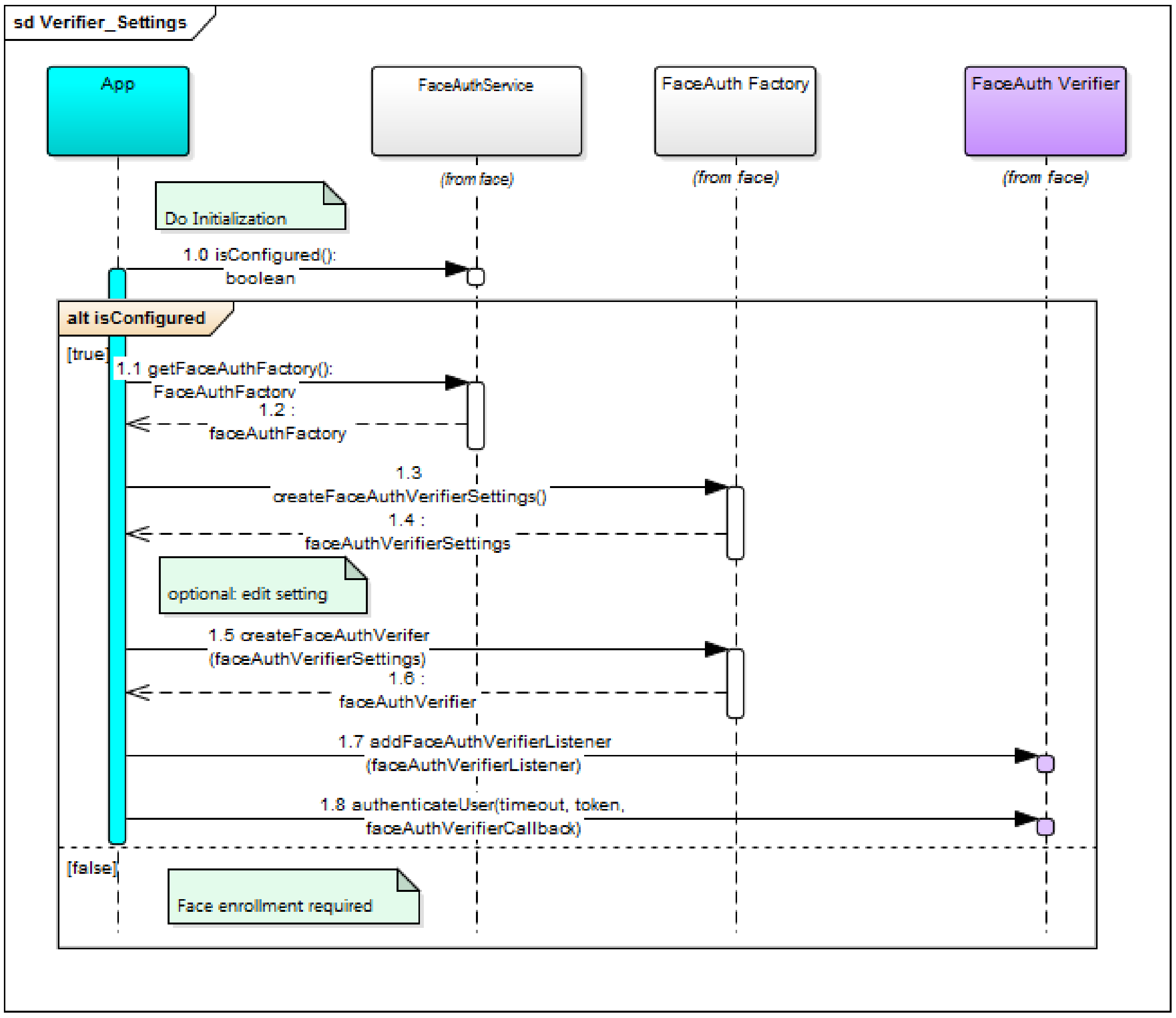
```
            }
            // check success
        }
    }
```

FaceID Verification

Creation of FaceAuthVerifier

`FaceAuthVerifier` is the interface for performing face verification. The following figure demonstrates the checks before setting up the verifier.

Figure 9 Creation of Verifier



sd Verifier_Settings
App
FaceAuthService
(from face)
FaceAuth Factory
(from face)
FaceAuth Verifier
(from face)
Do Initialization
1.0 isConfigured(): boolean
alt isConfigured
[true]
1.1 getFaceAuthFactory(): FaceAuthFactory
1.2 : faceAuthFactory
1.3 createFaceAuthVerifierSettings()
1.4 : faceAuthVerifierSettings
optional: edit setting
1.5 createFaceAuthVerifer (faceAuthVerifierSettings)
1.6 : faceAuthVerifier
1.7 addFaceAuthVerifierListener (faceAuthVerifierListener)
1.8 authenticateUser(timeout, token, faceAuthVerifierCallback)
[false]
Face enrollment required

On Android:

```
FaceAuthFactory faceAuthFactory = faceAuthService.getFaceAuthFactory();
FaceAuthVerifierSettings verifierSettings =
        faceAuthFactory.createFaceAuthVerifierSettings();
```

```
// Option 1: Create verifier with setting FaceAuthVerifier verifer =
         faceAuthFactory.createFaceAuthVerifer(verifierSettings);

// Option 2: Create default verifier FaceAuthVerifier defaultVerifier =
         faceAuthFactory.createFaceAuthVerifer();
```

On iOS:

```
// Create face auth factory
EMFaceAuthFactory* faceAuthFactory = [faceAuthService faceAuthFactory];


// Option 1: Create Verifier with setting
EMFaceAuthVerifierSettings *verifierSetting =
         [faceAuthFactory createFaceAuthVerifierSettings];
[verifierSetting setMatchingThreshold:customMatchingThreshold];
id<EMFaceAuthVerifier> verifier =    [faceAuthFactory
createFaceAuthVerifierWithSettings:verifierSetting];


// Option 2: Create default verifier id<EMFaceAuthVerifier> defaultVerifier =
         [faceAuthFactory createFaceAuthVerifierWithDefaultSettings];
```

Setting up Listener and Callbacks

A callback function need to be set in order to listen to the face verification event.
On Android:

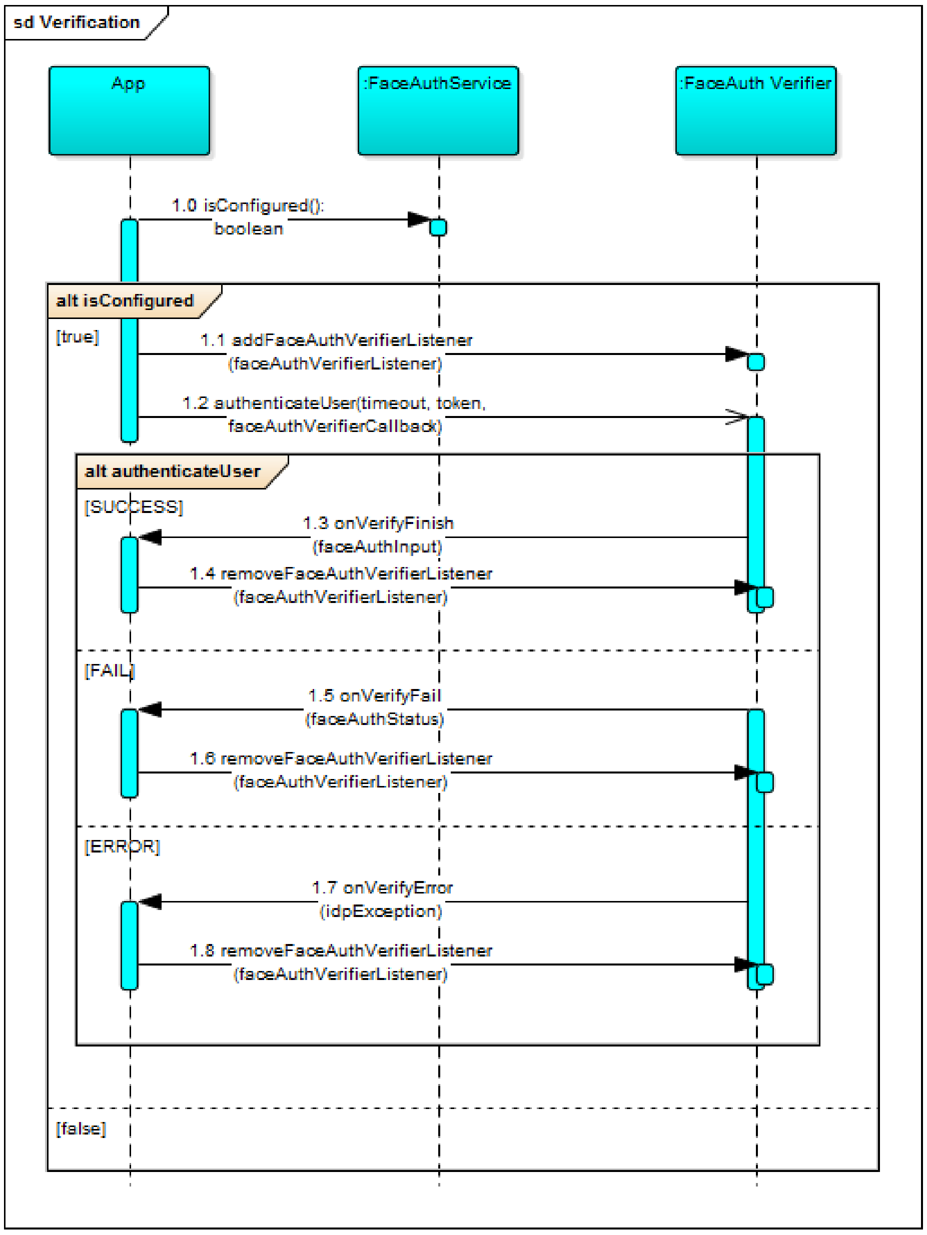
```
//create verifier
final FaceAuthVerifier verifier =
         faceAuthService.getFaceAuthFactory()
                       .createFaceAuthVerifer(verifierSettings);
final FaceAuthVerifierListener listener =
      new FaceAuthVerifierListener() {
         @Override
         public void onFrameReceived(FaceAuthFrameEvent event) {
                       }
 };
verifier.addFaceAuthVerifierListener(listener);
// timeout in millisecond
int timeout = 30000;

// Start to authenticate the user with face.
// The result will be returned from the listener callbacks.
verifier.authenticateUser(timeout, token,
                            new FaceAuthVerifierCallback() {
     @Override
     public void onVerifyFinish(FaceAuthInput faceAuthInput) {
         verifier.removeFaceAuthVerifierListener(listener);
     }
    @Override
    public void onVerifyFail(FaceAuthStatus status) {
         //handle error according to the status returned
         verifier.removeFaceAuthVerifierListener(listener);
     }
    @Override
    public void onVerifyError(IdpException e) {
         //handle error
         verifier.removeFaceAuthVerifierListener(listener);
      }
 });
```

On iOS:

```
id<EMFaceAuthVerifier> _faceAuthVerifier;


// The class "self" should implement
// protocol EMFaceAuthVerifierDelegate.
[_faceAuthVerifier setFaceAuthVerifierDelegate:self];

[_faceAuthVerifier authenticateUser:_tokenPp3
                          withTimeOut:60000
                    completionHandler:
     ^(id<EMFaceAuthInput> faceAuthInput, NSError *error) {


      if (error) {
              // verification failed, proceed to error handler.
     } else {
         // verification success, proceed to other steps,
         // e.g. OTP generation.
     }
     // Update UI, e.g. prompt some UI notifications.

    // Remove the delegate once the verification complete.
     [_faceAuthVerifier removeFaceAuthVerifierDelegate];
}];
```

The following diagram illustrates the process of setting up the verifier and listeners.

Figure 10 Verification Flow


sd Verification
App
:FaceAuthService
:FaceAuth Verifier
1.0 isConfigured(): boolean
alt isConfigured
[true]
1.1 addFaceAuthVerifierListener (faceAuthVerifierListener)
1.2 authenticateUser(timeout, token, faceAuthVerifierCallback)
alt authenticateUser
[SUCCESS]
1.3 onVerifyFinish (faceAuthInput)
1.4 removeFaceAuthVerifierListener (faceAuthVerifierListener)
[FAIL]
1.5 onVerifyFail (faceAuthStatus)
1.6 removeFaceAuthVerifierListener (faceAuthVerifierListener)
[ERROR]
1.7 onVerifyError (idpException)
1.8 removeFaceAuthVerifierListener (faceAuthVerifierListener)
[false]

Handling of the FrameEvent

The face verification is automatically handled by Ezio Mobile SDK. Ezio Mobile SDK will perform the verification based on the verification settings. Currently, there are two modes of verifications—IMAGE and LIVENESS_PASSIVE. IMAGE mode requires only one image to be extracted and verified. LIVENESS_PASSIVE mode requires the user to follow the instructions from Ezio Mobile SDK to perform liveness action checks.
Currently, Ezio Mobile SDK prompts the user to keep still or to blink based on the verification setting.
The application is called with a frame event whenever a frame has been captured by the Ezio Mobile SDK.

Visual Feedback

The frame event contains the image that can be directly displayed to the user to provide a visual feedback based on what has been captured by the camera. This listener/callback is similar as that of the enrollment process.
On Android:

```
@Override
public void onFrameReceived(FaceAuthFrameEvent event) {
     // always display the image to the user
     // to provide visual feedback
     // FaceView class is provided by the SDK.
     // It needs to be declared in your layout.xml
     faceView.setFaceFrameEvent(event);


     //Alternatively, you can used the default Android ImageView widget
     imageView.setImageBitmap(event.getImage().toBitmap());
    ...
}
```

On iOS:

```
// required function from EMFaceAuthVerifierDelegate protocal
// Handle verification process here
- (void)verifier:(id<EMFaceAuthVerifier>)verifier
    didUpdateFaceAuthFrameEvent:(id<EMFaceAuthFrameEvent>)frameEvent
{
     // Get the captured image to display on UIImageView.
     UIImage *image = [frameEvent image];
}
```

Response to EventFrame Information

The frame event also contains other information to allow the application to give instructions to the user to complete the verification process. This listener/callback is similar as that of enrollment process.

- Face coordinate
  The coordinates of the face with respect to the complete face image. It gives information about the size of the actual face as well as the position of the face. The application checks if the actual face is centered within the rectangle coordinates and if the size of the face is satisfying. Users may be prompted to center their faces or to move closer to the camera if needed.
  On Android:

```
@Override
public void onFrameReceived(FaceAuthFrameEvent event) {

 // check face coordinates
 Rect faceCoordinate = event.getFaceCoordinate();
 if(faceCoordinate.bottom == 0 && faceCoordinate.top == 0
  && faceCoordinate.left == 0 && faceCoordinate.right == 0){
        // face not detected, prompt the user
  }
  ...
}
```

On iOS:

```
// required function from EMFaceAuthVerifierDelegate protocal
// Handle verification process here
- (void)verifier:(id<EMFaceAuthVerifier>)verifier
   didUpdateFaceAuthFrameEvent:(id<EMFaceAuthFrameEvent>)frameEvent{

    // Application is advised to check the following attributes
    // before the frame is added.
    // (Recommended) check face coordinates.
     CGRect imageBound = [frameEvent imageBounds];
     if(imageBound.size.height==0 && imageBound.size.width==0){
         // Face not detected.
         return;
     }
 }
```

- Liveness score
  The score detected by Ezio Mobile SDK. This is linked to the configured liveness threshold. Liveness check is successful if the score exceeds the threshold. Face liveness detection is currently only applied in face verification.
  On Android:

```
@Override
public void onFrameReceived(FaceAuthFrameEvent event) {
       // check the status to prompt user in an error status
       FaceAuthStatus status = event.getStatus();
      // for LIVENESS mode, check the liveness that is being
      // asked by the SDK
       EnumSet<FaceAuthLivenessAction> liveness =
                                   event.getLivenessAction();
     if(liveness != null
         &&liveness.contains(FaceAuthLivenessAction.KEEP_STILL)){
        // ask user to keep still for a few seconds
        // the length of the duration depends on liveness
        // threshold setting
     }
      if (liveness != null
         && liveness.contains(FaceAuthLivenessAction.BLINK)){
         // ask user to blink
     }
     ...
}
```

On iOS:

```
// required function from EMFaceAuthVerifierDelegate protocal
// Handle verification process here
- (void)verifier:(id<EMFaceAuthVerifier>)verifier
 didUpdateFaceAuthFrameEvent:(id<EMFaceAuthFrameEvent>)frameEvent{

   // Extract frame information and notify user
```

```
      EMFaceAuthLivenessAction action =[frameEvent livenessAction];
      if(action != EMFaceAuthLivenessActionNoLiveAction){
            switch (action) {
              case EMFaceAuthLivenessActionBlink:
                   // ask user to blink
                   message = @"Blink required...";
                   break;
              case EMFaceAuthLivenessActionKeepStill:
                   // ask user to keep still for a few seconds
                   // the length of the duration depends on
                   // liveness threshold setting
                   message = @"Keep still...";
                   break;
               default:
                   break;
       }
   }
```

- Liveness score
  The score detected by Ezio Mobile SDK. This is linked to the configured liveness threshold. Liveness check is successful if the score exceeds the threshold.

Note:

Face liveness detection is currently only applied in face verification.

Verification Cancellation and Cleanup

The verification process must be canceled if the user decides to terminate the process while it is still in progress. Together with the cancellation, the frame event listener must be removed.
On Android:

```
// any time when user cancels during the verification,
// the following needs to be called
// and the listener MUST be removed
verifier.cancelVerificationProcess();
verifier.removeFaceAuthVerifierListener(faceAuthVerifierListener);
```

On iOS:

```
// any time when user cancels during the verification,
// the following needs to be called and the delegate MUST be removed


//cancel verification
[verifier cancel];


// Remove delegate method
[verifier removeFaceAuthVerifierDelegate]
```

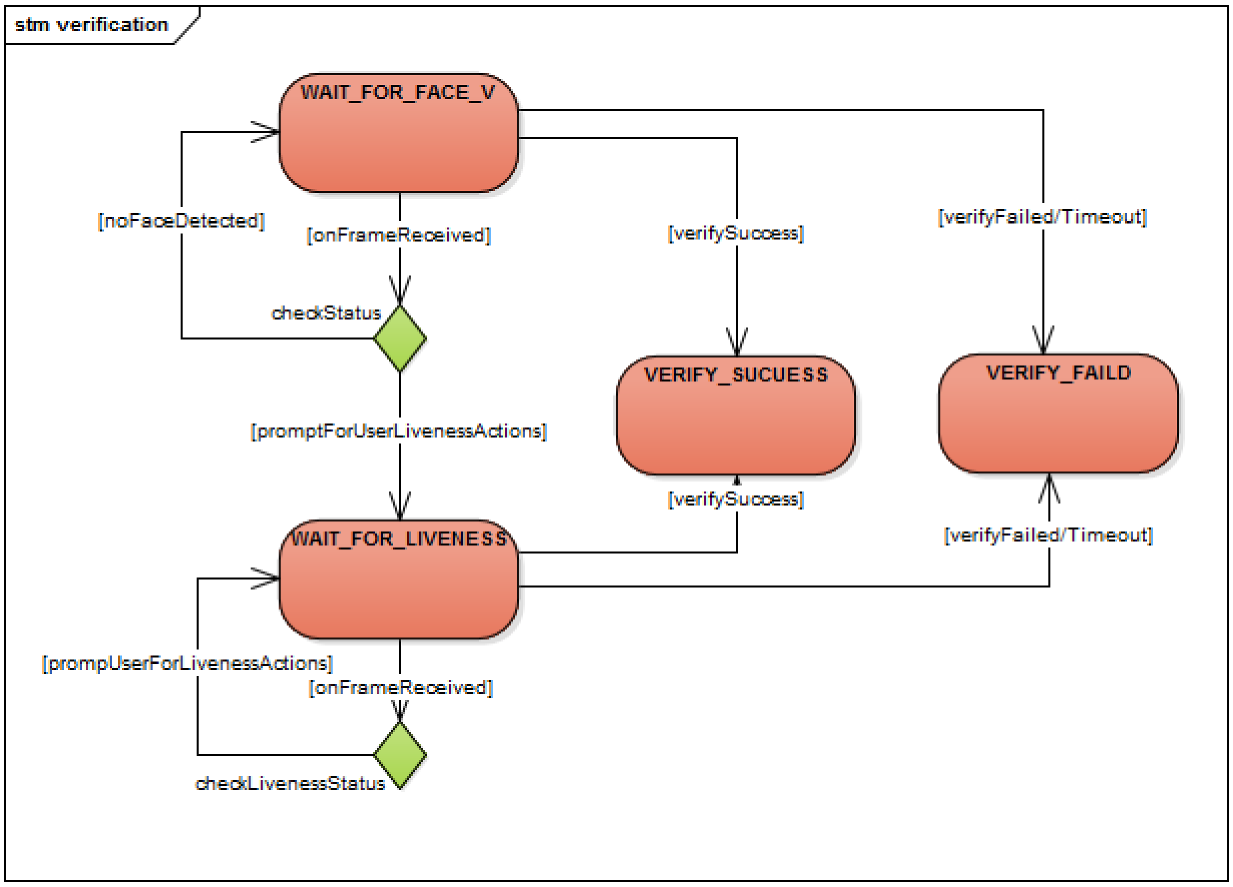
Note:

Failure to remove the listener will cause potential memory leaks and malfunctioning of the face authentication feature.

Verification Process

The follow figure illustrates the flow of a face verification process.

Figure 11 Face Verification Flow



To start the face verification process, it requires a human face to be first detected by Ezio Mobile SDK. If the IMAGE mode is used, the internal verification starts once a valid face is extracted. To determine if a face is to be marked for extraction, Ezio Mobile SDK checks the current face image captured by the camera and determines if it qualifies to be verified based on the internal algorithms. The background lighting and position of the face affect the extraction of the image. The verification settings such as quality threshold are taken into consideration when the internal algorithm is doing image extraction. On the application side, the application prompts the user to adjust the position of the face based on the face coordinates returned from the frame event.

In LIVENESS_PASSIVE mode, apart from extracting a valid image, Ezio Mobile SDK also requires some liveness actions to be performed by the user such as to keep still without blinking or moving. Subsequently, a blinking action is required by the user to complete the verification process.

Error Codes

This section contains a list of error codes that may occur on Android and iOS platforms during the facial authentication.

- On Android
  The following table contains the selected error codes and their corresponding exceptions. The full list of error codes can be found in the `FaceAuthResultCode` class.

  Table 1 Error Codes on Android

Error Code	Description
FACE_FRAME_CREATION_ERROR	The fingerprint authentication cannot be used due to non-enrolled fingerprints.
NO_FACE_REGISTERED	No faces are enrolled.
NO_CAMERA_AVAILABLE	No camera is detected on the device.
INVALID_CAMERA_NAME	The camera name provided is invalid.
ALREADY_INITIALIZED	The library has been initialized.

- iOS
  The following table contains the selected error error codes and their corresponding exceptions. The full list of error codes can be found in the `em_status.hclass`.

  Table 2 Error Codes on iOS

Error Code	Description
EM_STATUS_AUTHENTICATION_FACIAL_TIMEDOUT	The facial authentication timeout is reached.
EM_STATUS_AUTHENTICATION_FACIAL_BAD_QUALITY	The image quality is too low (too bright, too dark, or too blur) to perform the biometric action.
EM_STATUS_AUTHENTICATION_FACIAL_MATCH_NOT_FOUND	User is not recognized as an enrolled user.
EM_STATUS_AUTHENTICATION_FACIAL_NOT_SUPPORTED	The front camera is not detected on the device.
EM_STATUS_AUTHENTICATION_FACIAL_NOT_INITIALIZED	The face engine has not been initialized.
EM_STATUS_AUTHENTICATION_FACIAL_CANCELED	Biometric action has been canceled.

Enroll and Verify Settings

This section introduces the configurable parameters in face enrolment and verification settings. Ezio Mobile SDK has provided a set of default values, however developers could modify these parameters based on the needs.
The following code snippets demonstrate how to create and modify the settings:

- On iOS:

```
// Create enroller setting
 EMFaceAuthEnrollerSettings *enrollerSetting = [faceAuthFactory
createFaceAuthEnrollerSettings];;


// Modify enroller setting
[enrollerSetting setQualityThreshold:customQualityThreshold];
```

- On Android:

```
// Create enroller setting
FaceAuthEnrollerSettings enrollerSetting =
faceAuthService.getFaceAuthFactory().createFaceAuthEnrollerSettings();


// Modify enroller setting
enrollerSetting.setQualityThreshold(customQualityThreshold);
```

The following section introduces the configurable parameters. Please note there are inconsistencies between iOS and android APIs. The following table is based on iOS API. For android please refer to Android API doc.

- Shared settings
  The following parameters are shared between `FaceAuthEnrollerSettings` and `FaceAuthVerifierSettings`.

Table 3 Shared Settings

Parameter	Description	Value Range	Default Value
faceCaptureMode	The capture mode to use. Image mode should be used to enroll a user, liveness mode is highly recommended for user verification.	IMAGE, LIVENESS_PASSIVE	IMAGE for enrolment and LIVENESS_PASSIVE for verification
livenessTreshold	This threshold is used in liveness mode only and determine how long the 'still phase' will be and how strict the check of liveness will be.	[0, 100]	0
qualityThreshold	Quality of the face detection to be achieve. This value is used to reject bad quality images.	[0, 100]	48

- Enrolment Settings
  The following parameters belongs to `FaceAuthEnrollerSettings`.

Table 4 Enrolment Settings

Parameter	Description	Value Range	Default Value
countdownToCapture	Seconds before the capture starts. Set to 0 to skip the countdown phase.	>= 0	5
numberOfFramesToEnroll	Number of face frames to enroll.	>= 1	4

- Verification Settings
  The following parameters belongs to `FaceAuthVerifierSettings`.

Table 5 Verification Settings

Parameter	Description	Value Range	Default Value
livenessBlinkTimeout	Face liveness blink timeout in milliseconds. The face verifier will return an error if no eye blink is detected within the timeout during face verification.	[0,30000]	5000
matchingTreshold	Confidence of the matching. The higher the value is set, the lower the false acceptance rate (FAR) will be.	[0,72]	48

Security Considerations

When using FaceID, the attack surface of the mobile application is larger than that for biometrics fingerprint or PIN. Templates and assets linked to facial authentication are protected and stored locally on the file system, unlike biometrics fingerprint which uses a hardware-backed implementation or PIN which does not store any derived data. Therefore, the risks such as theft threats or forensic analysis are higher for this authentication method. Resistance of this authentication method relies mainly on application/software resistance.
Refer to *Ezio Mobile SDK V4.9 Security Guidelines* "GEN29 Matching Threshold and FAR/FRR for FaceID Authentication" in `EzioMobileSDK_security_guideline.pdf` for other details.

FaceID Reset

Note:

FaceID reset is only available on Android platform only.

This is used to remove the existing face template and deactivates Face Authentication Mode for existing tokens protected with Face Authentication Mode. The application user is required to re-enrol the face template and reactivates this authentication mode to protect the generation of one-time password.

On Android:

```
try {
 faceAuthService.reset();
} catch (IdpException e) {
// handle error
}
```

Migration for Android Q Upgrade

Google has deployed the Android Q Beta, and there are specific changes on data privacy enforcement that forbid collection of device information for third party applications.

With the new Android Q, some APIs' usage is restricted in getting the IMEI and device fingerprint source service details. As Ezio Mobile SDK is dependent on device information, it is important to migrate to the new changes which are independent of device information.

For more information on Android Q privacy changes, refer to https://developer.android.com/preview/privacy/data-identifiers#device-ids

---

Warning:

To ensure that active end users are not affected by the Android Q privacy changes, the following must be done:

- The app needs to be updated with Ezio Mobile SDK V4.9.2 to migrate the data related to Ezio features.
- Call the migration APIs in the app start up to make sure that the data is migrated.
- The application has to call these migration APIs before the device is upgraded to Android Q, since the migration requires the device information

If the migration is not done before the device is upgraded to Android Q, the Ezio Mobile SDK features will not work.

Migration Steps

For migration, there are two new APIs for each module.

- `isMigrationNeededForAndroidQ`
- `migrateForAndroidQ`

1. The first API `isMigrationNeededForAndroidQ` is used to check if the migration is needed. If this method returns a `true` value, the end user has to give the permission for READ_PHONE_STATE.

   If the `isMigrationNeeded` API returns a `false` value, the READ_PHONE_STATE permission will not be required. Hence, permission is not required for the new user.
2. Once the permission is given, the `migrateForAndroidQ` API need to be called to trigger the data migration for Android Q.

Code Snippets

The following code snippets are used for Android Q migration.

- Face Auth Module

```
FaceAuthService faceAuthService =
FaceAuthService.create(AuthenticationModule.create());


if (faceAuthService.isMigrationNeededForAndroidQ()) {
    //Ask for READ_PHONE_STATE permission


    //Once the user permissions are granted
    faceAuthService.migrateForAndroidQ();
}
```

Frequently Asked Questions

This chapter contains the commonly asked questions about Ezio Mobile SDK.

- Can you explain possible values and configuration of IPB?
  IPB comes from the CAP specification from MasterCard in the token compression step. The IPB applies to the assembled token data which consists of:

  ```
  PSN        ||   CID     ||    ATC     ||    AC      ||   IAD
  <-1 byte->  <-1 byte->  <-2 bytes->  <-8 bytes->  <-8 bytes->
  ```

  The default IPB value of the mobile SDK is:
  `00 00 00 FF FF FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00`
  This gives an effective length of 24 bits, resulting in a CAP token of up to 8 digits (refer to *Chip Authentication Program - Functional Architecture 2007* (MasterCard)). The following table lists the maximum token length, depending on the effective IPB length (the number of 1-bits in the IPB). An actual token may be shorter, depending on its IPB value.

  Table 6 Length of IPB and its Maximum Token Length

Effective IPB Length	Maximum Token Length
17 – 19	6
20 – 23	7
24 – 26	8
27 – 29	9
30 – 33	10
34 – 36	11
37 – 39	12
40 – 43	13
44 – 46	14
47 – 49	15
50 – 53	16

  Other CAP configurations may have an impact on the effective IPB length, for example, IAF bit 7 (2nd bit from the left) will decide if the 1st byte of IPB (the PSN) is used (b7 = 1) or omitted (b7 = 0).

- Does the SDK support client certificates for mutual authentication in the provisioning sequence?
  No, the SDK does not support this feature for provisioning. Therefore, the server's TLS software must be configured to accept clients without certificates.

- Why am I getting an exception when I generate OTP after updating my iOS app from App Store to my current version?
  Normally, updating the app on App Store works without a problem. However, the testing of update works

outside the App Store environment is a bit tricky and usually fails when generating OTP from a new version although it has been provisioned previously in the older version. This is because a fingerprint data key component known as `identifierForVendor` changes when updating an app installed from App Store to a newer version via other means such as Xcode or iTunes. This does not indicate a real failure of the client but one that requires a special procedure to test this use case, refer to the following steps to counter this update failure:
Procedure:

a. Build/resign your previous app version to update (App A) with an ad hoc provisioning profile.
b. Build your updated version (App B) with the same provisioning profile.
c. Use iTunes to import App A to iTunes Library by double clicking or a simple drag. Install your app on a device by selecting the connected device and navigating to **Apps** tab. Click **Install**, and sync.
d. Perform the provisioning, generate OTP and execute your tests.
e. Build and install Previous App Version.
f. Next import App B to iTunes library. This will replace the previous library.
g. Click **Update** and sync.
h. Perform the OTP generation.

- How is the EPS public key transferred to the mobile application?
  As it is the role of the mobile application to set the EPS public key. It means that the mobile application can decide to hard-code it if they want.
- Is there any sample challenge identifying different templates for Dynamic Signature?
  The template provides a list of primitives to be used in the computation of Dynamic Signature which can be retrieved from the challenge in the unconnected mode by the SDK. Certain combinations of the templates and sample challenges are listed as follows:

Table 7 Templates and Sample Challenges

Template ID	Challenge
Template 0	00007
Template 1	00024
Template 2	00048
Template 3	00069
Template 4	00082
Template 5	00105
Template 6	00122
Template 7	00146
Template 8	00167
Template 9	00180
Template 10	00200
Template 11	00228
Template 12	00244
Template 13	00263
Template 14	00285
Template 15	00302
Template 16	00325
Template 17	00341
Template 18	00360
Template 19	00387
Template 20	00409
Template 21	00421
Template 22	00445
Template 23	00466

- Why is the SDK not working after the restoration?
  There are two possible reasons that explain why the OTP verification no longer works after an application restoration. First, the application used fingerprint mechanism to seal the credentials with the device restoring application data does not work on a different device. It is suggested to perform a new provisioning in this case. Moreover, if the application data are restored on the same device assuming that the fingerprint data has not changed, there is still a chance that the OTP cannot be verified if the token used is an event-based type. The reason for this is that a backup/restore may de-synchronize the counter. In this case, there is no way to re-synchronize it as accepting the previous OTPs is considered as a major security issue.

- For iOS, why am I getting "duplicate symbols" linker error when I use Reachability or KeychainItemWrapper class distributed by Apple?
  One of the components of Ezio Mobile SDK uses the popular utility classes `Reachability` and `KeychainItemWrapper` which result in duplicate symbols. As a temporary workaround, you can perform any of the following methods:

- Rename your `Reachability`/`KeychainItemWrapper` classes. Apple grants you to use, modify or reproduce in source or binary form with or without modification. For more information on the license, refer to:
    - {+}https://developer.apple.com/library/ios/samplecode/Reachability+
    - {+}https://developer.apple.com/library/ios/samplecode/GenericKeychain+
- Remove the implementation files (`.m`) of these classes.

Document History

This document starts from 4.0 release.

Version (X.yy)	Date (dd/mm/yyyy)	Changes
4.3	30/01/2017	First version of FaceID authentication mode PG.
4.6	16/01/2018	Update the instructions of native libraries installation and configurations of obfuscation.

Terminology

This section contains abbreviations and terms found in this document or related documents.

Abbreviations

EPS	Enrollment Provisioning Server
ESN	Electronic Serial Number
HOTP	HMAC based One Time Password
HSM	Hardware Security Module
IMEI	International Mobile Equipment Identifier
IMSI	International Mobile Subscriber Identifier
OATH	Open Authentication
OOB	Out Of Band
OCRA	OATH Challenge-Response Algorithm
OTP	One Time Password
PM	Password Manager
PIN	Personal Identification Number
PTC	PIN Try Counter
RC	Registration Code
SM	Security Module (see also HSM and SSM)
SMS	Short Message Service
SSM	Software Security Module
TLS	Transport Layer Security
TOTP	Time-based One Time Password
URL	Universal Resource Locator
VIC	Verify Issuer Code
VICATC	VIC Application Transaction Counter
VICTC	VIC Try Counter
DSKPP	Dynamic Symmetric Key Provisioning Protocol

Glossary of Terms

Provisioning Access Domain is a SIM Toolkit/UICC Toolkit parameter that specifies the identities or access rights (CHV & ADM) granted to an application to access GSM/UICC files and perform actions on these files. For example, if the Access Domain value is "FF" (No Access to the File System), attempts to access a file cause an exception.

Token The Ezio Mobile SDK's representation of a user's credentials.

User The mobile device user.