VisionLabs LUNA ID

v.1.20.0

Table of contents

1. Introduction

LUNA ID POS is a software development kit (SDK) designed for integration into Point-of-Sale (POS) terminals. It enables face detection, image quality estimations, and subsequent biometric data transmission for processing, The SDK integrates LUNA PLATFORM 5 for advanced processing, OneShotLiveness estimation, and identity matching.

Start here

- Licensing
- Initial setup
- Support & resources
- Download docs

Getting the best shot

- Best shot estimations
- Getting the best shot

OneShotLiveness

- Offline OneShotLiveness
- Online OneShotLiveness

Interaction with LUNA PLATFORM

- Overview
- Configuration

2. General info

2.1 Overview

LUNA ID POS is a software development kit (SDK) designed for integration into Point-of-Sale (POS) terminals. It enables face detection, image quality estimations, and subsequent biometric data transmission for processing, The SDK integrates LUNA PLATFORM 5 for advanced processing, OneShotLiveness estimation, and identity matching.

2.1.1 Key features

LUNA ID POS provides the following features:

- Getting the best shot:
  - Estimating the best shot by the following criteria:
    - Number of faces in the frame
    - Face detection bounding box size
    - Frame edges offset
    - Eye state (open, closed, or occluded)
    - Head pose (pitch, yaw, and roll)
    - AGS (Average Garbage Score, frame quality check for subsequent descriptor extraction)
    - Image quality (lightness, darkness, and blurriness)
    - Face occlusion
  - Submitting the best shot with the detected face to LUNA PLATFORM 5 to perform OneShotLiveness estimation on the backend
  - Submitting the best shot with the detected face to LUNA PLATFORM 5 for backend descriptor matching
  - Interface customization (displaying face functionality in end-client applications)

2.2 System requirements

2.2.1 Hardware requirements

The following minimal hardware specifications are required for LUNA ID POS operation:

Component	Requirement
CPU architecture	arm64-v8a, armeabi-v7a
RAM	Minimum 2 GB
Camera	Front-facing camera with minimum 720p resolution recommended

2.2.2 Software requirements

Component	Requirement
Android API Level	21 or higher
Android NDK	Version 28 or higher
Operating system	Android 5.0 or higher

2.3 Technical support and resources

If you have questions, problems or just need help with LUNA ID POS, you can either contact our Technical Support or try to search for the needed information using other help resources.

You can contact our Technical Support via email:

- support@visionlabs.ru

3. Licensing

3.1 Activating the license

To integrate LUNA ID POS with your project and use its features, you need to activate the license.

The license activation mechanism is as follows:

LUNA ID POS first checks if you provided a license file via the initEngine method.

If provided, the license is directly passed to the engine.

If not provided, the system attempts to read the license from the assets folder and passes it to the engine.

If no license is found in either location, the activation process fails.

To activate the license, follow the step below:

1. Obtain the following parameters from VisionLabs:

Parameter	Description
`Server`	The URL of the license server.
`EID`	A unique identifier for your application.
`ProductID`	The product identifier for LUNA ID POS.

2. Specify the parameters in *license.conf* and save the changes.

Example structure of *license.conf*

Here is an example structure of the file:

```
<?xml version="1.0"?>
<settings>
   <section name="Licensing::Settings">
         <param name="Server" type="Value::String" text="https://example-
license-server.com"/>
         <param name="EID" type="Value::String" text="your-eid-here"/>
         <param name="ProductID" type="Value::String" text="your-product-id-
here"/>
         <param name="Filename" type="Value::String" text="license.dat"/>
         <param name="ContainerMode" type="Value::Int1" x="0"/>
         <param name="ConnectionTimeout" type="Value::Int1" x="15"/>
         <param name="licenseModel" type="Value::Int1" x="2" />
   </section>
</settings>
```

3. Save the *license.conf* file in the *assets/data/license.conf* directory of your project.

The license key will be generated and saved to the specified directory. The license file has a binary format. At the next launch of your app on the same device, the license will be read from this file.

4. Call the `initEngine()` method to initialize LUNA ID POS and activate the license.

Here is an example implementation:

```
private fun initLunaSdk() {
   val baseUrl = "url"
   val token = "token"
   val headers = mapOf("Authorization" to token)
   val apiHumanConfig = ApiHumanConfig(baseUrl, headers)
   val lunaConfig = LunaConfig.create(
      acceptOccludedFaces = true,
      acceptOneEyed = false,
      acceptEyesClosed = false,
      detectFrameSize = 350,
      skipFrames = 36,
      ags = 0.5f,
      bestShotInterval = 500,
      detectorStep = 1,
      usePrimaryFaceTracking = true,
```

```
        glassesChecks = setOf(GlassesCheckType.GLASSES_CHECK_SUN)
    )

    FacePay.initEngine(
        app: Application,
        lunaConfig: LunaConfig,
        apiHumanConfig: ApiHumanConfig? = null,
        license : File? = null,
        timeoutMillis : Long = 30_000L
    )
}
```

Note

The parameters in the example are set to default values. Adjust them according to your requirements.

Key components of the example code

The example code has the following components:

Component	Description
`baseUrl`	A variable that specifies the URL to LUNA PLATFORM 5. For details, see Interaction with LUNA PLATFORM 5.
`token`	A variable that specifies a LUNA PLATFORM 5 token, which will be transferred to a request header from LUNA ID POS.
`headers`	A map that specifies headers that will be added to each request to be sent to LUNA PLATFORM 5.
`apiHumanConfig`	An optional configuration parameter for calling the LUNA PLATFORM 5 API. Can be set to `null` if no LUNA PLATFORM 5 API calls are required. This will also disable the Online OneShotLiveness estimation, regardless of the `onlineLivenessSettings` argument.
`ApiHumanConfig`	A class required for configuration to call the LUNA PLATFORM 5 API.
`lunaConfig`	An argument to be passed for best shot parameters.
`LunaConfig`	A class that describes best shot parameters.
`acceptOccludedFaces`	A parameter that specifies whether an image with an occluded face will be considered the best shot. For details, see Getting the best shot with an occluded face.
`acceptEyesClosed`	A parameter that specifies whether an image with two closed eyes will be considered the best shot. For details, see Getting the best shot with faces with closed eyes.
`detectFrameSize`	A parameter that specifies a face detection bounding box size.
`skipFrames`	A parameter that specifies a number of frames to wait until a face is detected in the face recognition area before video recording is stopped.
`ags`	A parameter that specifies a source image score for further descriptor extraction and matching. For details, see AGS.
`bestShotInterval`	A parameter that specifies a minimum time interval between best shots.
`detectorStep`	A parameter that specifies a number of frames between frames with full face detection.
`usePrimaryFaceTracking`	A parameter that specifies whether to track the face that was detected in the face recognition area first.
`glassesChecks`	A parameter that specifies what images with glasses can be best shots. For details, see Getting the best shot with faces with occluded eyes.
`FacePay.initEngine`	A method that activates the LUNA ID POS license.
`license`	An instance of **java.io.File**. If this parameter is not provided, the system will use the default *license.conf* file located in the project.

Component	Description
`timeoutMillis`	The timeout for license activation, with a default value of 30 seconds (30,000 milliseconds).

5. Subscribe to events from the LunaID.engineInitStatus flow to monitor the initialization process:

```
LunaID.engineInitStatus.flowWithLifecycle(this.lifecycle, Lifecycle.State.STARTED)
.onEach {
    if(it is LunaID.engineInitStatus.InProgress) {
   // LUNA ID POS is loading
   }else if(it is LunaID.engineInitStatus.Success) {
   // LUNA ID POS is ready
    }
}.flowOn(Dispatchers.Main)
.launchIn(this.lifecycleScope)
```

3.2 License parameters

The table below outlines the parameters required for license activation and subsequent processing in LUNA ID POS:

Parameter	Required	Default value	Description
`Server`	✓	Not set	The URL of the activation server used to validate and activate the license.
`EID`	✓	Not set	A unique identifier (Entitlement ID) assigned to your application.
`ProductID`	✓	Not set	The specific product identifier for LUNA ID POS.
`Filename`		`license.dat`	The default name of the file where the activated license is saved. Maximum length: 64 characters. Changing this name is not recommended.
`ContainerMode`		0	Indicates whether the application is running in a containerized environment.
`ConnectionTimeout`		15	Specifies the maximum time (in seconds) allowed for the license activation request. Setting this value to 0 disables the timeout. Negative values are not allowed. Maximum value: 300 seconds.
`licenseModel`		2	Defines the license to be used. Possible values: • `1` - Thales • `2` - Zeus

4. Initial setup

This section provides step-by-step instructions for setting up and integrating LUNA ID POS into Android applications.

4.1 Step 1: Download LUNA ID POS

LUNA ID POS is distributed as a collection of modular archive files containing essential libraries and neural networks for mobile application integration.

1. Obtain the distribution kit from VisionLabs.
2. Move the archives to the desired directory.

4.2 Step 2: Configure repository

Add the following repository configuration to the *settings.gradle.kts* file:

> **Info**
>
> The *settings.gradle.kts* file is located in the root directory of your project and defines which projects and libraries you need to add to the classpath of your build script.

```
dependencyResolutionManagement {
    repositories {
        google()
        mavenCentral()

        // VisionLabs repository configuration
        ivy {
            url = java.net.URI.create("https://download.visionlabs.ru/")
            patternLayout {
                artifact("releases/lunaid-[artifact]-[revision].[ext]")
                setM2compatible(false)
            }
            credentials {
                username = getLocalProperty("vl.login") as String
                password = getLocalProperty("vl.pass") as String
            }
            metadataSources { artifact() }
        }
    }
}
```

4.3 Step 3: Provide user credentials

In the *local.properties* file, specify your credentials:

```
vl.login=YOUR_LOGIN
vl.pass=YOUR_PASSWORD
```

Implement the following utility function to access credentials programmatically:

```
fun getLocalProperty(key: String, file: String = "local.properties"): Any {
    val properties = java.util.Properties()
    val localProperties = File(file)

    if (localProperties.isFile) {
        InputStreamReader(FileInputStream(localProperties), Charsets.UTF_8).use {
reader ->
            properties.load(reader)
        }
    } else {
        error("File not found: '$file'")
    }

    if (!properties.containsKey(key)) {
        error("Key '$key' not found in file '$file'")
    }

    return properties.getProperty(key)
}
```

Warning

Add *local.properties* to your *.gitignore* file to prevent credential exposure through version control systems.

4.4 Step 4: Specify dependencies

Add the necessary *.aar* files as dependencies in your module's *build.gradle.kts*. This file defines various build parameters including dependencies, plugins, library versions, compilation options, and testing configurations.

Add the following dependencies to your *build.gradle.kts*:

```
dependencies {
    // CameraX dependencies (required for camera functionality)
    implementation(libs.camera.core)
    implementation(libs.camera.camera2)
    implementation(libs.camera.lifecycle)
    implementation(libs.camera.video)
    implementation(libs.camera.view)

    // VL-LUNA-ID-POS core modules
    implementation("ai.visionlabs.lunaid:core:${DepVersions.sdkVersion}@aar")
    implementation("ai.visionlabs.lunaid:common-arm:${DepVersions.sdkVersion}
@aar")
    implementation("ai.visionlabs.lunaid:cnn60-arm:${DepVersions.sdkVersion}@aar")
    implementation("ai.visionlabs.lunaid:mask-arm:${DepVersions.sdkVersion}@aar")
    implementation("ai.visionlabs.lunaid:glasses-arm:${DepVersions.sdkVersion}@aar")
    implementation("ai.visionlabs.lunaid:oslm-arm:${DepVersions.sdkVersion}@aar")
    implementation("ai.visionlabs.lunaid:security:${DepVersions.sdkVersion}@aar")
    implementation("ai.visionlabs.lunaid:facePayCore:${DepVersions.sdkVersion}
@aar")

}
```

4.5 Step 5: Initialize LUNA ID POS

Initialize the SDK once during application startup:

```
@HiltViewModel
class MainViewModel @Inject constructor(
    val application: Application,
    val dataStore: AppDataStore,
    private val userDao: UserDao
) : ViewModel() {

    init {
        // Initialize the SDK engine
        initEngine()

        // Optional: Monitor initialization status
        monitorEngineStatus()
    }

    private fun initEngine() = CoroutineScope(Dispatchers.Default).launch {
        // Configure recognition parameters
        val config = LunaConfig(livenessType = LivenessType.Offline)
```

```
        // Configure server connection for data processing
        val apiHumanConfig = ApiHumanConfig(
            baseUrl = "https://luna-api-aws.visionlabs.ru/6/",
            headers = mapOf(
                "Authorization" to "Bearer <YOUR_JWT_TOKEN>"
            )
        )

        // Load license file from application's internal storage
        val customLicense = File(application.filesDir, "license.conf")

        // Initialize VL-LUNA-ID-POS
        FacePay.initEngine(application, config, apiHumanConfig, customLicense)
    }

    private fun monitorEngineStatus() {
        LunaID.engineInitStatus
            .onEach { status -> Log.d("FacePay", "engineInitStatus=$status") }
            .launchIn(CoroutineScope(Dispatchers.IO))
```

```
    }
}
```

Component	Type	Description
`MainViewModel`	Class	Primary `ViewModel` hosting initialization logic.
`init {...}`	Initialization block	Executes when the `MainViewModel` instance is created. Calls `initEngine()` and sets up initialization status monitoring.
`initEngine()`	Function	Contains core LUNA ID POS initialization logic.
`LunaID.engineInitStatus`	Flow	Provides subscription to LUNA ID POS initialization state changes for process monitoring.
`Log.d("FacePay", ...)`	Logging	Outputs current initialization status to system log for debugging.
`LunaConfig`	Class	Defines LUNA ID POS operational parameter configuration.
`livenessType`	Parameter	Sets OneShotLiveness estimation mode. `LivenessType.Offline` indicates local device processing without server communication.
`ApiHumanConfig`	Class	Configures interaction with LUNA PLATFORM 5 API.
`baseUrl`	Variable	Specifies LUNA PLATFORM 5 URL for request submission (for example, verification requests).
`headers`	Map	Defines headers appended to each request sent to LUNA PLATFORM 5.
`customLicense`	Variable	File object pointing to license file (`license.conf`) in application's private storage.
`FacePay.initEngine(...)`	Method	Primary LUNA ID POS initialization method. Accepts application context, configuration objects, and starts the recognition engine.

4.6 Step 6: Launch the camera

Initiate face capture by calling the `FacePay.showCamera` method:

```
@OptIn(ExperimentalCamera2Interop::class)
fun openCamera(context: Context) {
    val settings = dataStore.settings().first()

    val showCameraParams = settings.showCameraParams.copy(
```

```
        borderDistanceStrategy =
BorderDistancesStrategy.WithCustomView(R.id.faceCaptureOverlay),
        checkSecurity = true // Virtual camera detection
    )

    FacePay.showCamera(
        context = context,
        params = showCameraParams,
        interactions = Interactions.Builder().build(),
        commands = Commands.Builder().build()
    )
}
```

4.7 Step 7: Get the best shot

Subscribe to the `bestShot` flow to capture best shots:

```
LunaID.bestShot
    .filterNotNull()
    .onEach { bestShot ->
        // Extract processed (warped) face image
        val faceBitmap = bestShot.bestShot.warp

        // Example: Send image to external system (Rosreestr in this case)
        sendToExternalSystem(faceBitmap)
    }
    .launchIn(viewModelScope)
```

5. Working with LUNA ID POS

5.1 Best shots

5.1.1 About best shot estimations

This section explains how LUNA ID POS evaluates image quality to get the best shot from a video stream.

How it works

LUNA ID POS analyzes each frame of a video stream captured by your device's camera, searching for a face. For accurate evaluation, each frame must contain only one face. Frames with faces that pass specific estimations are considered the best shots.

If an estimation fails, the corresponding error message is returned.

The minimum camera resolution required for optimal estimator performance is 720p (1280x720 pixels).

The `LunaID.allEvents()` event (or the more specialized `LunaID.finishStates()`) emits the `ResultSuccess` event containing the best shot found and an optional path to the recorded video.

You can adjust parameters for best shot estimations in *LunaConfig.kt*.

Estimations

LUNA ID POS performs several estimations to determine if an image qualifies as the best shot.

NUMBER OF FACES IN THE FRAME

The estimation ensures that the frame contains only one face. If multiple faces are detected, the system returns a `TooManyFacesError` error message.

By default, no value is set for this estimation.

AGS ESTIMATION

The estimation calculates a score indicating the suitability of the source image for descriptor extraction and matching. The output is a normalized float score ranging from 0 to 1. A score closer to 1 indicates better matching results for the image.

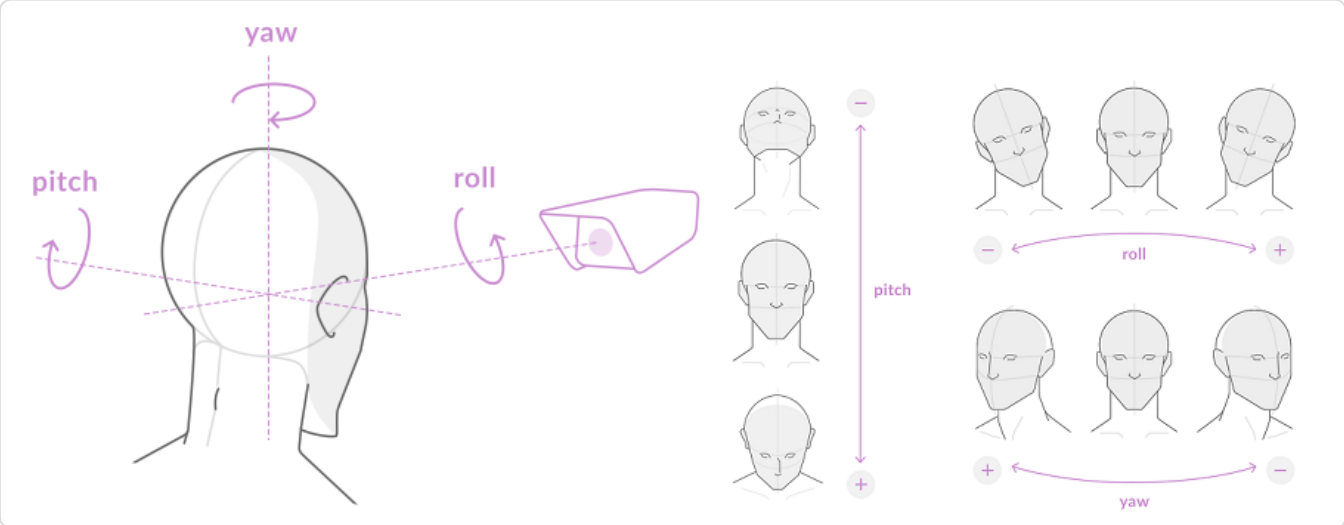
By default, the AGS threshold is set to 0.5.

Implementation: `public val ags: Float = DEFAULT_AGS`

HEAD POSE ESTIMATION

The estimation determines a person's head rotation angles in 3D space, specifically along the pitch, yaw, and roll axes:

- **Pitch (X-axis)**: This angle measures the vertical tilt of the head. It limits the head rotation along the X-axis.
- **Yaw (Y-axis)**: This angle measures the horizontal rotation of the head. It limits the head rotation along the Y-axis.
- **Roll (Z-axis)**: This angle measures the lateral tilt of the head. It limits the head rotation along the Z-axis.



Head pose

By default, all rotation angles (pitch, yaw, and roll) are set to 25 degrees each side.

Implementation:

Angle	Implementation
Pitch	`public val headPitch: Float = DEFAULT_HEAD_PITCH`
Yaw	`public val headYaw: Float = DEFAULT_HEAD_YAW`
Roll	`public val headRoll: Float = DEFAULT_HEAD_ROLL`

IMAGE QUALITY ESTIMATION

The estimation evaluates an image based on several key criteria to ensure it meets the necessary standards. These criteria include:

- **Blurriness**: The image appears out of focus.
- **Underexposure**: The image is too dark.
- **Overexposure**: The image is too bright.

Below are the default values for each criterion used in the image quality estimation:

Parameter	Default value
Blurriness	0.61
Lightness	0.57
Darkness	0.50

FACE DETECTION BOUNDING BOX SIZE

The estimation ensures that the detected face's bounding box matches a specified size. This estimation helps determine if the subject is too far from the camera, affecting image quality.

The minimum recommended size for the face bounding box is 200 x 200 pixels.

The default value is 350 dp (density-independent pixels).

> **Warning**
>
> If the converted pixel value is less than 100 pixels, the frame size will automatically be set to 100 pixels to maintain a minimum acceptable quality.

Here are the configuration details for setting the minimum detectable frame size: `public const val DEFAULT_MIN_DETECT_FRAME_SIZE: Int = 350`

Implementation: `public val detectFrameSize: Int = DEFAULT_MIN_DETECT_FRAME_SIZE`

FRAME EDGES OFFSET

The estimation calculates the distance from the detected face's bounding box to the edges of the image.

Minimal border distance:

- Without OneShotLiveness estimation: The minimal border distance for best shot estimation is 0 pixels. This means the face can be right at the edge of the frame.
- With OneShotLiveness estimation: The minimal border distance increases to 10 pixels to ensure sufficient space around the face for accurate OneShotLiveness estimation.

The default value is set to 0 pixels.

Implementation: `public val borderDistance: Int = DEFAULT_BORDER_DISTANCE`

EYE STATE

The estimation determines whether the eyes in a detected face are open or closed.

The estimation is performed only if eye interaction is enabled.

FACE OCCLUSION

The estimation determines whether the lower part of the face in the frame is occluded by an object. This feature allows you to define whether such frames can still be considered as best shots.

You can enable or disable the estimation via the `LunaConfig.acceptOccludedFaces` parameter. By default, this parameter is set to `true`, meaning that no estimations for occluded faces are performed.

```
val config = LunaConfig.create(
            ...
            acceptOccludedFaces = true
            ...
        )
```

When `acceptOccludedFaces = false`, LUNA ID POS checks for occlusions of the nose, mouth, and lower part of the face. If an occlusion is detected, it triggers the `OccludedFace` error.

If `acceptOccludedFaces` or `acceptMask` are set to `true`, LUNA ID POS skips the corresponding estimations for face occlusions or medical masks, respectively.

Implementation: `public val acceptOccludedFaces: Boolean = true`

MEDICAL MASK ESTIMATION

The estimation determines whether the face in a frame is partially covered by a medical mask. This feature allows you to define whether such frames can still be considered as best shots.

If `acceptOccludedFaces` or `acceptMask` are set to `true`, LUNA ID POS skips the corresponding estimations for face occlusions or medical masks, respectively.

By default, `acceptMask` is set to `true`, allowing frames with occluded faces to be considered as potential best shots. Adjust this setting based on your specific requirements.

Implementation: `public val acceptMask: Boolean = true`

GLASSES ESTIMATION

The estimation determines whether the eyes in a frame are occluded by glasses.

You can specify detailed rules for eye occlusion:

- Images of people wearing sunglasses *cannot* be considered best shots.
- Images of people wearing eyeglasses *cannot* be considered best shots.
- Images of people wearing any type of glasses *cannot* be considered best shots.

For details, see Getting the best shot with faces with occluded eyes.

5.1.2 Getting the best shot

With LUNA ID POS, you can capture video stream and get the best shot on which the face is fixed in the optimal angle for further processing.

> **Tip**
>
> In LUNA ID POS you can specify a face recognition area for best shot selection.

Step 1: Start the camera

Call the `FacePay.showCamera()` method to start the camera session. This method initiates face detection and analysis within the video stream.

Step 2: Get the list of best shots (optional)

Implement the step, if you want to get multiple best shots during a session. You can then send the list of acquired best shot to the backend for estimation aggregation.

Set the `LunaConfig.multipartBestShotsEnabled` parameter to true to get multiple frames.

Specify the number of best shots to be returned by setting the `LunaConfig.bestShotsCount` parameter. The valid range of values for `bestShotsCount` is from 1 to 10.
When `multipartBestShotsEnabled` is active, the list of best shots will be returned in the `BestShotsFound` event. Use the `bestShots` Flow to collect this list.

Structure of `BestShotsFound`:

```
data class BestShotsFound(
    val bestShots: List<BestShot>?
) : Event()
```

Usage example:

```
LunaID.bestShots.filterNotNull().onEach { bestShotsList ->
    Log.e(TAG, "bestShots: ${bestShotsList.bestShots}")
}.launchIn(viewModelScope)
```

This Flow continuously gets a list of best shots as they are detected during the session.

Step 3: Subscribe to the final best shot result

To retrieve the final best shot result (including metadata such as `videoPath` and `interactionFrames`), subscribe to the `LunaID.bestShot` Flow.

Structure of `BestShotFound`:

```
data class BestShotFound(
    val bestShot: BestShot, // The selected best shot
    val videoPath: String?, // Path to the recorded video (if enabled)
    val interactionFrames: List<InteractionFrame>? // Frames with Dynamic Liveness
interactions (optional)
) : Event()
```

Usage example:

```
val bestShotFlow = MutableStateFlow<Event.BestShotFound?>(null)

LunaID.bestShot.filterNotNull().onEach { bestShotFound ->
    Log.e("BestShotFound", bestShotFound.toString())
    // Process the best shot or its associated metadata here
}.launchIn(viewModelScope)
```

Step 4: Handle best shot events

The system gets events for both individual best shots (`BestShotFound`) and lists of best shots (`BestShotsFound`). Depending on your use case, handle these events accordingly:

Event	Description
`BestShotFound`	Contains the final best shot and optional metadata. Use this for single-best-shot scenarios.
`BestShotsFound`	Contains a list of all best shots detected during the session. Use this for multi-best-shot scenarios.

Face recognition area

In some cases, you may need the best shot search to start only after a user places their face in a certain area in the screen. You can specify face recognition area borders by implementing one of the following strategies:

- Border distances are not initialized
- Border distances are initialized with an Android custom view
- Border distances are initialized in dp
- Border distances are initialized automatically

Add a delay before starting face recognition

You can optionally set up a fixed delay or specific moment in time to define when the face recognition will start after the camera is displayed in the screen. To do this, use the `StartBestShotSearchCommand` command.

Add a delay before getting the best shot

You can optionally set up a delay, in milliseconds, to define for how long a user's face should be placed in the face detection bounding box before the best shot is taken. To do this, use the `LunaID.foundFaceDelayMs` parameter. The default value is 0.

5.1.3 Getting the best shot with an occluded face

In LUNA ID POS, you can define whether images with occluded faces can be considered as best shots. This feature allows you to customize the behavior based on your specific requirements.

To determine whether an image with an occluded face will be considered the best shot, use the `LunaConfig.acceptOccludedFaces` parameter.

The `acceptOccludedFaces` parameter has the following values:

Value	Description
`true`	Default. An image with an occluded face can be considered the best shot.
`false`	An image with an occluded face cannot be considered the best shot. The `BestShotsFound` event will appear in `LunaID.bestShots()` with payload `DetectionError.OccludedFace` every time an occluded face is recognized.

Warning

The `acceptOccludedFaces` parameter requires the *lunaid-mask-X.X.X.aar* dependency.

To define that images with occluded faces can be considered as best shots:

1. Add the required *.plan* files to your project dependencies:

```
implementation("ai.visionlabs.lunaid:mask:X.X.X@aar")
```

2. Specify the `acceptOccludedFaces` parameter in `LunaConfig`:

```
LunaConfig.create(
    acceptOccludedFaces = true
)
```

5.1.4 Getting the best shot with faces with closed eyes

In LUNA ID POS, you can define whether images with faces with one or two closed eyes can be considered best shots.

One closed eye

To get the best shot with a closed eye, use the acceptOneEyeClose parameter. The parameter has the following values:

Value	Description
`true`	Default. Specifies that frames that contain faces with a closed eye can be best shots.
`false`	Specifies that frames that contain faces with a closed eye cannot be best shots. However, it is possible to get the best shot with an occluded eye. For details, see Getting the best shot with faces with occluded eyes.

Warning

The `acceptOneEyeClose` parameter requires the `acceptOneEyed` parameter to be enabled.

Two closed eyes

To get the best shot with two closed eyes, use the `acceptEyesClosed` parameter. The parameter has the following values:

Value	Description
`true`	Specifies that frames that contain faces with closed eyes can be best shots.
`false`	Default. Specifies that frames that contain faces with closed eyes cannot be best shots.

Consider an example below:

```
LunaConfig.create(
acceptEyesClosed = false,
)
```

5.1.5 Getting the best shot with faces with occluded eyes

In LUNA ID POS, you can define whether an image in with occluded eyes can be considered the best shot.

You can specify the following eye occlusion rules:

- Images of people in sunglasses cannot be best shots.
- Images of people in eyeglasses cannot be best shots.
- Images of people in any glasses cannot be best shots.

To get best shots with faces with occluded eyes:

1. Add the required .plan files to the dependency:

```
implementation("ai.visionlabs.lunaid:glasses:X.X.X@aar")
```

2. Specify the `glassesChecks` parameter in `LunaConfig` to define the type of glasses in the image and whether the image can be the best shot:

```
lunaConfig = LunaConfig.create(
        glassesChecks = setOf(GlassesCheckType.GLASSES_CHECK_SUN,
GlassesCheckType.GLASSES_CHECK_DIOPTER)
    )
```

The `glassesChecks` parameter specifies what images with glasses can be best shots.

Possible values:

Value	Description
`GlassesCheckType.GLASSES_CHECK_SUN`	Defines that images with people in sunglasses cannot be best shots.
`GlassesCheckType.GLASSES_CHECK_DIOPTER`	Defines that images with people in eyeglasses cannot be best shots.

You can specify either one, none, or both possible values.

The default value is not set.

5.1.6 Errors

The table below lists best shot errors:

Error	Description
PrimaryFaceLostCritical	The primary face that was detected in the video stream has been lost.
PrimaryFaceLost	The primary face was not detected in the video stream or has been lost.
FaceLost	Unable to detect a face in the video stream.
TooManyFaces	The frame must contain only one face for LUNA ID POS to perform a series of estimations, and then select the best shot.
FaceOutOfFrame	A face is too close to the camera and does not fit the face recognition area.
FaceDetectSmall	The size of the detected face does not correspond to the specified bounding box size size.
BadHeadPose	Head rotation angles are not between the minimal and maximum valid head position values.
BadQuality	The input image does not meet the AGS estimation threshold.
BlurredFace	The input image does not meet the blurriness threshold.
TooDark	The input image does not meet the darkness threshold.
TooMuchLight	The input image does not meet the lightness threshold.
GlassesOn	The person in the input image is wearing sunglasses.
OccludedFace	The face is not properly visible in the input image.
BadEyesStatus	The eye state estimation failed.
FaceWithMask	The person in the input image is wearing a medical mask.

5.2 OneShotLiveness

5.2.1 About OneShotLiveness estimation

OneShotLiveness is an algorithm for determining whether a person in one or more images is "real" or a fraudster using a fake ID (printed face photo, video, paper, or 3D mask).

OneShotLiveness is used as a pre-check before performing face detection.

OneShotLiveness estimation types

With LUNA ID POS, you can perform the following types of OneShotLiveness estimation:

- **Online OneShotLiveness estimation**
  To perform Online OneShotLiveness estimation, LUNA ID sends a request to the LUNA PLATFORM 5 `/liveness` endpoint. For more details about LUNA ID and LUNA PLATFORM 5 interaction, see the Interaction of LUNA ID with LUNA PLATFORM 5.
- **Offline OneShotLiveness estimation**
  To perform Offline OneShotLiveness estimation, you do not need to send requests to LUNA PLATFORM 5. You can perform the estimation directly on your device.

Image requirements

An image that LUNA ID takes as input must be a source image and meet the following requirements:

OneShotLiveness thresholds

By default, two thresholds are used for OneShotLiveness estimation:

- Quality threshold
- Liveness threshold

QUALITY THRESHOLD

Quality threshold estimates the input image by the following parameters. The table below has the default threshold values. These values are set to optimal:

Parameter	Threshold	Value
Blurriness	`blurThreshold`	0.61
Darkness (underexposure)	`darknessThreshold`	0.50
Lightness (overexposure)	`lightThreshold`	0.57
Illumination	`illuminationThreshold`	0.1
Specularity	`specularityThreshold`	0.1

LIVENESS THRESHOLD

The `LunaConfig.livenessQuality` parameter specifies the threshold lower which the system will consider the result as a presentation attack.

For images received from mobile devices, the default liveness threshold value is **0.5**. For details, see Liveness threshold.

Number of best shots

You can specify a number of best shot to be collected for a OneShotLiveness estimation. To do this, use the `LunaConfig.bestShotsCount` parameter.

The default value is 1.

5.2.2 Performing Online OneShotLiveness estimation

You can automatically perform Online OneShotLiveness estimation by sending a request to the LUNA PLATFORM 5 `/liveness` endpoint. The estimation allows you determine if the person in the image is a living person or a photograph. You can then validate the received images with LUNA PLATFORM 5.

To perform Online OneShotLiveness estimation:

1. Specify the `livenessType: LivenessType` field in `LunaConfig`. The field accepts one of the following values:

Value	Description
`None`	Disables the estimation. The default value.
`Online`	Enables the estimation by sending a request to the LUNA PLATFORM 5 `/liveness` endpoint.

2. Specify the required LUNA PLATFORM 5 server parameters in `ApiHumanConfig`.

The example below shows how to enable Online OneShotLiveness estimation:

```
val apiConfig = ApiHumanConfig("http://luna-platform.com/api/6/")
    LunaID.init(
        ...
        apiHumanConfig = apiConfig,
        lunaConfig = LunaConfig.create(
         livenessType = LivenessType.Online,
        ),
    )
```

5.2.3 Performing Offline OneShotLiveness estimation

With LUNA ID POS, you can perform liveness estimation directly on your device. Unlike Online OneShotLiveness estimation, which sends requests to the LUNA PLATFORM 5 `/liveness` endpoint, Offline OneShotLiveness estimation operates locally, ensuring faster processing and reduced dependency on backend services.

This feature allows you to determine whether the person in the image is a living individual or a spoof (for example, a photograph or mask).

To perform Offline OneShotLiveness estimation:

1. Add the required dependency.

Add the appropriate dependency to your *build.gradle* file based on your device's architecture. This dependency includes the neural networks required for Offline OneShotLiveness estimation.

```
implementation("ai.visionlabs.lunaid:oslm-arm:X.X.X@aar")
```

2. Specify the estimation type in `LunaConfig`:

```
LunaConfig.create(
   livenessType = LivenessType.Offline
)
```

3. Specify the neural networks to be used for the estimation by using the `LunaConfig.livenessNetVersion` parameter. This parameter is of type `LivenessNetVersion` and supports two values:

Value	Description
`LITE`	Default. Loads the neural network models: • *oneshot_rgb_liveness_v12_model_4_arm.plan* • *oneshot_rgb_liveness_v12_model_5_arm.plan*
`MOBILE`	Loads only the *oneshot_rgb_liveness_v12_model_6_arm.plan* model. Recommended for devices with lower performance.

Warning

After changing the `livenessNetVersion` parameter, restart the final application for the changes to take effect.

```
LunaConfig.create(
    livenessType = LivenessType.Offline,
    livenessNetVersion = LivenessNetVersion.LITE
)
```

Logging

When configuring the `livenessNetVersion` parameter, you can now monitor which networks are loaded directly from the logs:

- `livenessNetVersion = 1` - The system loads: *oneshot_rgb_liveness_v12_model_6_arm.plan*
- `livenessNetVersion = 2` - The system loads: *oneshot_rgb_liveness_v12_model_4_arm.plan* and *oneshot_rgb_liveness_v12_model_5_arm.plan*

5.2.4 Disabling OneShotLiveness estimation

If you want to skip a liveness estimation over the best shot, you can disable a OneShotLiveness estimation.

To disable OneShotLiveness estimations, set the `livenessType: LivenessType` field to `None` in `LunaConfig`.

If `livenessType: LivenessType` is not specified, OneShotLiveness estimations are disabled by default.

The example below shows how to disable OneShotLiveness estimations:

```
val apiConfig = ApiHumanConfig("http://luna-platform.com/api/6/")
    LunaID.init(
        ...
        apiHumanConfig = apiConfig,
        lunaConfig = LunaConfig.create(
         livenessType = LivenessType.None,
        ),
    )
```