SAFR®

from realnetworks

SAFR® Android SDK Documentation

Documentation Version = 3.030

Publish Date = October 6, 2021

Contents

1 SAFR SDK Overview

1.1 Purpose

The SAFR SDK is a set of shared libraries that together implement an object tracking mechanism, face recognition and event reporting. The object tracker is used to locate and track different types of objects (for example, human faces and badges) in a video file or live video stream. Face recognition can learn new identities or match existing identities. The event reporter can notify your application of detection and recognition events. Object detection, face recognition, tracking and event reporting are performed in real time.

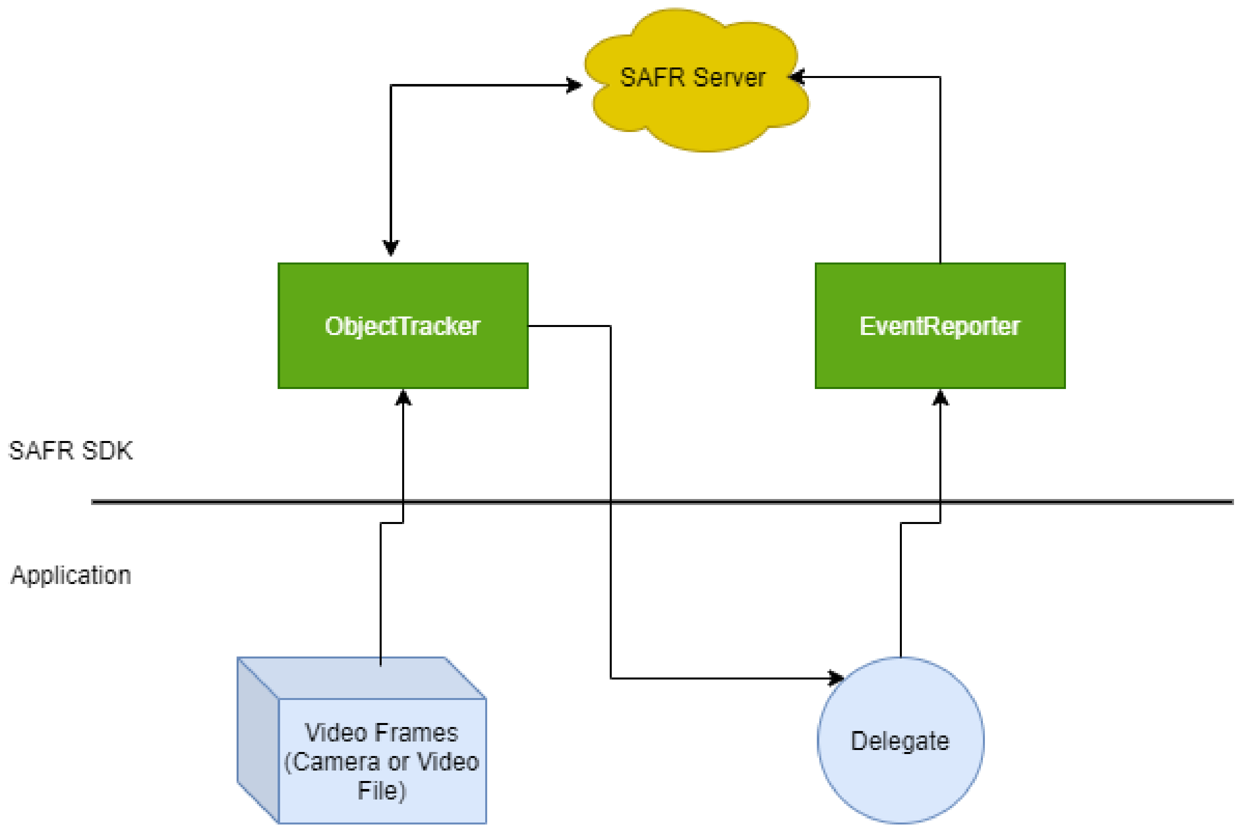
The SDK features a C API that exposes a set of ArgusKit objects. All SDK-defined functions and type definitions are prefixed with the `ARK` namespace qualifier. The SDK is distributed as a `tar.gz` file containing all necessary headers, documentation, and shared libraries. It also includes samples showing how the SDK can be used from your own application, and how the SDK components (shared libraries and auxiliary files) must be packaged with your application in order to ensure ArgusKit works correctly with your application.

While the SDK API design and architecture is the same across all supported platforms, the implementation language of the API may be different. For example, the API may be a C API for some platforms, while it is a Java or C# API for other platforms. Also, the packaging of the SDK is different for each platform to accommodate the technical requirements and limitations of the platform.

1.2 SDK Organization

The SAFR SDK defines a number of objects that an application uses to track faces or badges. The application creates these objects, configures them, and then interacts with them. Once the application no longer needs a SAFR SDK object, it is expected to release (*free*) the SAFR SDK objects. All SAFR SDK objects use reference counting to manage the lifetime of the objects.

The following diagram shows the two most important SAFR SDK objects: the `ObjectTracker` and the `EventReporter`, and how an application interacts with them:



The application receives video frames from a camera or a video file and hands them off to an ObjectTracker. The ObjectTracker processes the video frames internally to detect and track objects. The ObjectTacker sends the video frames to a SAFR Server (whether in the cloud or installed on-premise) for recognition. The ObjectTracker continuously informs the application about the current state of the ObjectTracker by invoking the ObjectTracker delegate. This delegate is provided by the application and receives information about objects found in the video input stream, objects that have disappeared, and objects that have changed their state. For example, an object may have been detected in one video frame, and it may have been recognized as a specific person in a later video frame. The ObjectTracker informs the application about this change of state.

An application is free to use the information about a tracked object any way it likes. It could display this information in a video overlay in its UI or it could process the information to generate statistical information or events. The SAFR SDK comes with an EventReporter able to process a stream of tracked objects to detect certain kinds of events that are then automatically reported to a SAFR Server.

Note: For SAFR SDK system requirements, go to SAFR.com.

2 Android SDK Installation

Create a SAFR account and download SAFR Android SDK from the Download Portal.

Note: The SAFR Android SDK is distributed as an Android library file (`aar`) which should be added as a library dependency to your Android project.

3 Use the Demo App for Android

The SDK ships with a simple example application demonstrating how to set up an object tracker, how to feed it with video frames from a camera, and how to receive information about detected and recognized persons from the object tracker.

Note: This demo application requires an active SAFR Cloud account in order to function in online mode. Search for the `USER_IDENTIFIER` and `PASSWORD` words in the source and replace them with your user identifier and password, or enter credentials via sign-in dialog.

The demo app features include:

- Camera feed processing — Face tracking, face detection, and face recognition.
    - Frame overlay is drawn on top of the tracked face; the level of details depends on current settings.
        - Age, gender, and sentiment info are drawn on top of the overlay.
        - Name and the face-quality metrics are drawn below the overlay.
        - Frame color depends on the person's attributes.
        - Heart icon is displayed when detected face is confirmed to be alive; a broken heart icon when detected face is confirmed not to be alive. (i.e. The subject failed an RGB liveness detection test.)
    - Face image sharing — Tapping the tracked face displays the native share picker; the shared image includes the extracted face.
- Photo import — Face detection and face recognition.
    - Selected photos are processed and displayed in another dialog.
    - Frame overlays are drawn on top of the face.
        - Frame color is set at random.
        - The face-quality metrics are displayed in the bottom-right corner of the image.
        - If recognized, the person's name is displayed below the frame.
    - Image sharing — Clicking the share button displays the native share picker; the original image with rendered overlay is shared.
    - Person (re)name — Tap the face to display the rename dialog.
- Configuration.
    - License - Dialog to enter the SDK license provided by RealNetworks. (Note that RGB liveness detection won't work without the SDK license.)
    - Sign-in — Allows entering SAFR credentials. Pressing **Skip** automatically transitions to Offline mode.
    - Camera facing - Front or back camera.
    - Face recognizer settings — Select or clear to enable or disable specific features:
        - Identity recognition.
        - Age recognition.
        - Gender recognition.
        - Mask recognition.
        - Occlusion recognition.
        - Sentiment recognition.
        - If all the face recognizer setting features are unchecked, the SDK automatically transitions to Offline mode.
    - Detector input size.
    - RGB liveness detector.
        - Minimum required face size.
        - Liveness detection threshold.
        - Fake detection threshold.
        - Detection scheme.
        - Frames evaluation count.
        - Minimum confirmations required.
    - Event reporter.
        - Event reporter switch.

        - RGB Liveness action - When enabled, events will be generated indicating detection of liveness for specific recognized persons.
    - User interface.
        - Show face info/quality attributes.
- Background service for RTSP stream processing - Enabled from the code.
- Run RTSP camera stream - Enabled from the code.

4 IDE and System Requirements

4.1 Android Studio 3.3

- https://developer.android.com/studio

4.2 Windows

- Microsoft Windows 7/8/10 (32- or 64-bit)
- 3 GB RAM minimum, 8 GB RAM recommended, plus 1 GB for the Android Emulator
- 2 GB of available disk space minimum, 4 GB Recommended (500 MB for IDE + 1.5 GB for Android SDK and emulator system image)
- 1280 x 800 minimum screen resolution

4.3 macOS

- macOS X 10.10 (Yosemite) or higher, up to 10.13 (macOS High Sierra)
- 3 GB RAM minimum, 8 GB RAM recommended, plus 1 GB for the Android Emulator
- 2 GB of available disk space minimum, 4 GB Recommended (500 MB for IDE + 1.5 GB for Android SDK and emulator system image)
- 1280 x 800 minimum screen resolution

4.3.1 Linux

- GNOME or KDE desktop
    - Tested on Ubuntu 14.04 LTS, Trusty Tahr (64-bit distribution capable of running 32-bit applications)
- 64-bit distribution capable of running 32-bit applications
- GNU C Library (glibc) 2.19 or later
- 3 GB RAM minimum, 8 GB RAM recommended, plus 1 GB for the Android Emulator
- 2 GB of available disk space minimum, 4 GB Recommended (500 MB for IDE + 1.5 GB for Android SDK and emulator system image)
- 1280 x 800 minimum screen resolution

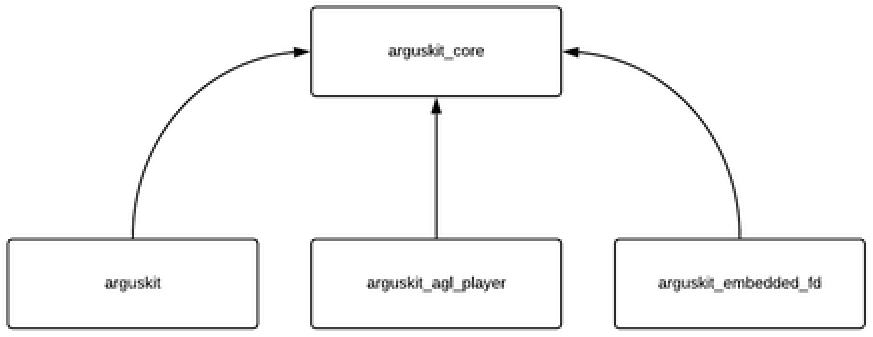
4.4 Supported Device Architecture

- ARM (armeabi-v7a, arm64-v8a)

4.5 Minimum Android Version

- Lollipop 5.x (API level 21)

5 SDK Structure



Modules are packed and distributed as standalone .aar libraries.

5.1 Arguskit core - arguskit_core.aar

- This is the root library in the dependency hierarchy - other modules depend on this.
- Includes interfaces and base classes needed by other modules.

5.2 Arguskit - arguskit.aar

Depends on *arguskit_core.aar*.

It provides following SAFR features bundled in a simple library:

- Face recognition using SAFR Platform.
- APIs for characterization and identity repository and management.
- ImageAnalyzer that executes face detection and recognition on images.
- CameraX and VisionCameras sources - Plug & play camera implementations.
- Entry points for using other modules.

5.3 Arguskit Embedded FD (Face Detector) - arguskit_embedded_fd.aar

Depends on *arguskit_core.aar*.

SAFR face detector implementation wrapped under the **EmbeddedFaceDetectorService** class. It features *Standard* and *High Sensitivity* (default) service configurations.

The number of processors depends on the device hardware capabilities and can be configured via the constructor. (The recommended number of processors is one.)

Implements the **FaceDetectorService** interface which must be passed into **Arguskit** via an **ObjectTracker.setFaceDetectorService(FaceDetectorService faceDetector)** call.

5.4 Arguskit AGL Player - arguskit_agl_player.aar

Depends on *arguskit_core.aar*.

Provides an rtsp:// player implementation. It uses the Android Mediacodec decoder.

It can be used to simply play local video files.

The integration is simple; SDK clients can add **AglVideoView** into their XML view hierarchy, provide a ‘rtsp://’ link or a local video filepath, and play it.

Produced video frames are accessed through a **AglVideoViewDelegate** callback and can be fed directly into **ObjectTracker**.

6 Android Studio Setup

The easiest way is to use the demo app as a project stub and continue development from there.

To manually set up your project to use the SAFR SDK, do the following:

1. Add dependencies to *arguskit_core.arr*, *arguskit.arr*, and *arguskit_embedded_fd.arr*. (File names may differ based on the library version.)

    1. Add libraries to your project:

        1. Click **File** > **New** > **New Module**.
        2. Click **Import .JAR/.AAR Package**, then click **Next**.
        3. Enter the location of the AAR file then click **Finish**.

    2. Make sure libraries are listed at the top of your *settings.gradle* file:

        - include `:app`, `:arguskit_core`, `:arguskit`, `:arguskit_embedded_fd`

    3. Open the app module's *build.gradle* file and add a new line to the dependencies block as shown in the following snippet:

        ```
        dependencies {
            implementation project(":arguskit_core")
            implementation project(":arguskit")
            implementation project(":arguskit_embedded_fd")
        }
        ```

    4. Click **Sync Project with Gradle Files**.

2. Update *AndroidManifest.xml*.

    ```
    <!-- Camera is required feature -->
    <uses-feature android:name="android.hardware.camera" />

    <!-- Add extra permissions -->
    <uses-permission android:name="android.permission.CAMERA" />
    <!-- Cloud communication -->
    <uses-permission android:name="android.permission.INTERNET" />
    <!-- Local video playback using AGL player -->
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"
       />
    ```

3. Add other dependencies to your app's *build.gradle*:

    ```
    dependencies {
        implementation project(":arguskit_core")
        implementation project(":arguskit")
        implementation project(':arguskit_embedded_fd')
        ...
        // Use the latest versions of these

        // Needed for VisionCameraSource
        implementation
           "com.google.android.gms:play-services-vision:${googlePlayServicesVersion}"

        // Needed for CameraXSource
        implementation "androidx.camera:camera-camera2:$cameraxVersion"
        implementation "androidx.camera:camera-lifecycle:$cameraxVersion"
        implementation "androidx.camera:camera-view:$cameraxViewVersion"
    ```

```
    // Serialization
    implementation "com.google.code.gson:gson:${gson}"

    // Network stack
    implementation "com.squareup.okhttp3:okhttp:${okhttp}"
}
```

4. Optionally include *arguskit_agl_player.aar*.
    1. Repeat steps from 1.
    2. Switch to **AglVideoView** and feed frames into **ObjectTracker**:

```
aglView.setVisibility(View.VISIBLE);

AglOptions options = new AglOptions();

//options.rotationOption = AglOptions.ROTATION_90;// Update rotation if
   needed

final String uri = RTSP_URI; // Or local video file path

aglView.init(uri, options, new AglVideoView.AglVideoViewDelegate() {

            @Override
            public void onFrameSizeChanged(int w, int h) {
                Log.d(TAG, "Stream resolution " + w + "x" + h + "
                   pixels");
            }

            @Override
            public void onFrameUpdate(VideoFrame videoFrame) {
                if (objectTracker != null) {
                    objectTracker.trackObjects(videoFrame);
             }
            }

            @Override
            public void onPreviewSizeChanged(float xScale, float yScale,
               int offsetLeft, int offsetTop) {
                // Tell the overlay that screen geometry has changed
                if (overlay != null) {
                    overlay.setCameraInfo(xScale, yScale, offsetLeft,
                       offsetTop,
                       CameraCharacteristics.LENS_FACING_FRONT);
                }
            }

            @Override
            public void onPlayerStreamEnd() {
                Log.d(TAG, "onPlayerStreamEnd ");
            }

            @Override
            public void onPlayerError(int error) {
                Log.e(TAG, "onPlayerError " + error);

            }
        }
);

aglView.start();
```

7 Configure the Android SDK

To configure the Android SDK, do the following:

1. Initialize settings, select cloud environment, and initialize SDK objects.

```
// Specify the cloud environment log-in credentials
// NOTE: replace USER_IDENTIFIER and PASSWORD with the user name and
   password that you have received from RN.
private static final String USER_IDENTIFIER = "obtain_from_RN";
private static final String PASSWORD = "obtain_from_RN";
private static final String DIRECTORY = "main";

// Set current mode/preset
private final ModeSettings.Identifier modeIdentifier =
   ModeSettings.Identifier.recognition;

@Override
public void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    //....

    // Create the instance
    settings = new Settings(getActivity().getApplication());
    //

        // Configuration
//        settings.setUserIdentifier(USER_IDENTIFIER);
//        settings.setUserPassword(PASSWORD);
//        settings.setUserDirectory(DIRECTORY);
//
   settings.setCloudEnvironment(CloudEnvironment.fromId(CloudEnvironment.ID_PROD))

        // To set a custom environment configuration

//        URL coviServerUrl = new URL("https://...");
//        URL eventServerUrl = new URL("https://...");
//        URL objectServerUrl = new URL("https://...");
//        URL rncvServerUrl = new URL("https://...");
//        URL virgaServerUrl = new URL("https://...");
//        settings.setCloudEnvironment(new
   CloudEnvironment(CloudEnvironment.ID_CUST, coviServerUrl,
   eventServerUrl, objectServerUrl, rncvServerUrl, virgaServerUrl));
}

@Nullable
@Override
public View onCreateView(@NonNull LayoutInflater inflater, @Nullable
   ViewGroup container, @Nullable Bundle savedInstanceState) {

        //...

        /**
         * Initialize Object tracker
         */
        objectTracker = new
           ObjectTracker(settings.objectTrackerConfiguration(modeIdentifier),
           getActivity(), settings);
        objectTracker.setDelegate(this);

        /**
         * Initialize and set the face detector service.
         */
        FaceDetectorConfiguration faceDetectorConfiguration =
           settings.faceDetectorConfiguration(modeIdentifier);
        faceDetectorConfiguration.license = "YOUR_SAFR_LICENSE"; // If
           not set, attempts to use the RGB liveness detection feature
           will result in RuntimeException
        //faceDetectorConfiguration.detectorModel =
           FDRuntimeOptions.FD_STANDARD; // Change detector model if
           needed
        objectTracker.setFaceDetectorService(new
           EmbeddedFaceDetectorService(getContext(),
           faceDetectorConfiguration, FD_INSTANCES_COUNT));

        /**
         * Initialize RGB liveness action recognizer
         */
        livenessActionRecognizer = new
           LivenessActionRecognizer(settings.livenessActionRecognizerConfiguration
        livenessActionRecognizer.setDelegate(this);

        /**
         * Initialize event reporting pipeline
         */
        final ObjectEventDataStore dataStore = new
           ObjectEventDataStore(); // Create the event date store
        objectEventLog = new ObjectEventLog(dataStore,
           ObjectEventLog.Context.live); // Initialize the objectEventLog
           with the data store with live context
        peopleIndexer = new
           PeopleIndexer(settings.peopleIndexerConfig(modeIdentifier),
           objectEventLog, dataStore); //Initialize the peopleIndexer
           which orchestrates event processing


        /**
         * To play & process rtsp streams use aglView
         */
        //initAglView();

        /**
         * Or play camera view.. check code for implementation
         */
        initCameraPreview();

    return view;
}
```

```
// Start/Stop peopleIndexer from onResume/OnPause callbacks

@Override
public void onResume() {
    super.onResume();
    peopleIndexer.startIndexing();

}

@Override
public void onPause() {
    super.onPause();
    peopleIndexer.stopIndexing();
}
```

2. Create the camera source and pass video frames to the object tracker.

```
/**
 * Creates the camera.
 */
private void createCameraSource() {
    //...
    argusCameraSource = new CameraXSource(getActivity(),
        cameraXSourcePreview);
    // Set ArgusCameraSourceCallback callback
    argusCameraSource.setSourceCallback(this);

    // Open camera with specific facing and profile
    argusCameraSource.openCamera(facing, cp);

    //...
}
```

   The SDK provides CameraXSource and VisionCameraSource sources but the client app can provide its own camera implementation.

3. Hook camera to **ObjectTracker** in the callback.

```
@Override
public void cameraSourceOnFrameUpdate(@org.jetbrains.annotations.Nullable
   VideoFrame frame) {
    // Pass camera frames to the objectTracker - never pass directly
        because the objectTracker uses its own threads.
    videoFrame.reset();
    videoFrame.shallowCopy(frame);

    if (objectTracker != null) {
         objectTracker.trackObjects(videoFrame);
    }
}

@Override
public void cameraSourceOnGeometryChanged(float xScale, float yScale, int
   offsetLeft, int offsetTop, int cameraFacing) {
    // Pass camera geometry info to a view which draws face info - this
       is essentially a mapping from camera to view coordinates
    if (overlay != null) {
        overlay.setCameraInfo(xScale, yScale, offsetLeft, offsetTop,
            cameraFacing);
    }
}

@Override
public void cameraSourceOnCameraReady(@org.jetbrains.annotations.Nullable
   CameraProfile cameraProfile) {
    // Tell the tracker to actually begin tracking once the camera is
       ready
    if (objectTracker != null) {
        objectTracker.beginTracking();
    }
}
```

The **startCameraSource()** method should be invoked from Activity's **onResume()**.

4. Process the results.

```
@Override
public void objectTrackerDidTrack(@NonNull ObjectTracker objectTracker,
   @NonNull TrackingResult trackingResult) {

    // Process the tracking result

    // Collect result
    result.clear();
    result.addAll(trackingResult.getAppeared());
    result.addAll(trackingResult.getDisappeared());
    result.addAll(trackingResult.getLingering());
    result.addAll(trackingResult.getUpdated());

    // Pass it to the overlay for display purposes
    overlay.setTrackedObjects(result);

    // Pass the tracking result to objectEventLog which logs and reports
       events to the cloud
    if (objectEventlog != null) {
        objectEventlog.update(trackingResult);
    }

    // Pass the tracking result to liveness recognizer
    if (livenessActionRecognizer != null) {
        livenessActionRecognizer.update(trackingResult);
    }
}
```

8 Android SDK Objects

8.1 Cloud Environment and Settings

This section describes the SAFR SDK objects which are used to control the SDK, which SAFR Cloud environment and account should be used by the object recognition functionality implemented by the SDK.

8.1.1 Settings and ModeSettings

Stores the global, SDK-shared configuration into the **SharedPreferences**. Note that this object is just a store; new settings are not applied automatically.

For every SDK object, there is an appropriate factory method which returns its configuration. So when a setting is changed, any potentially affected SDK object should be re-created. (In some cases only their configuration should be updated.)

A client app can register itself as a callback that is invoked when a single or bulk settings change is made, at which point the client app should re-create objects or update their configurations.

The difference between **Settings** and **ModeSettings** is that the former is the same across all modes, whereas the latter may be configurable per mode/preset.

8.1.2 ModeSettings.Identifier

A configuration preset or mode for specific SDK use cases. i.e. recognition - a configuration optimised for simple face recognition without event reporting.

8.1.3 CloudEnvironment

The **CloudEnvironment** object encapsulates the Cloud service to which the **ArgusKit** should connect to execute cloud-based functions like face recognition.

8.1.4 CloudAccount

The **CloudAccount** object stores the account name and password for the cloud account that should be used for cloud-based functionality like face recognition.

In most cases, it should not be accessed directly but should instead be accessed via the **Settings** object.

8.2 Tracking Objects in a Video

This section describes the SDK objects which you use to detect and recognize objects. The most important SDK object in this category is the object tracker.

The object tracker allows you to detect and recognize faces, get information about them, and it takes care of tracking their location and identity over time.

8.2.1 ObjectTracker.Configuration

An object tracker configuration stores all required and optional settings for an object tracker instance. You create a configuration instance and you fill it in with the desired options before you create an object tracker.

8.2.2 ObjectTracker

The object tracker is the heart of ArgusKit. It receives a stream of video frames which it analyzes to find objects. It tracks found objects as long as they remain visible to the video stream. Object detection, recognition, and tracking are executed in real-time. An object tracker is connected to a delegate which you pass to the object tracker at creation time. The **ObjectTracker** informs its delegate at every frame about the current state of the objects which it is tracking. The delegate can then use the tracked object APIs to learn what kind of objects the tracker found and what their current spatial location, size, and state are.

The **objectTrackerDidTrack()** callback is invoked on every frame update. The result of processing a given frame is wrapped within a **TrackingResult** object as a list of **TrackedObject**s.

In essence, **TrackedObject** just holds the details about the object being tracked; at this moment, it will simply represent a face. In other words, to differentiate what has really changed within a given frame **TrackingResult** provides API for:

- **trackingResult.getAppeared()** - The list of faces that appeared for the first time in this frame.
- **trackingResult.getDisappeared()** - The list of faces that were tracked in prior frames but that disappeared in this frame.
- **trackingResult.getLingering()** - The list of faces that are no longer detected but we're still tracking them and which may go away or come back soon. (You can configure lingering allowance via settings.)
- **trackingResult.getUpdated()** - The list of faces that changed their state in this frame.

8.2.3 TrackingResult

A **TrackingResult** object contains a snapshot of the current state of the object tracker. The object tracker delivers a new tracking result at every frame boundary. The tracking result contains a list of tracked object which have appeared in the current frame, which have disappeared and which have changed their current state in the current frame. For example, if a tracked object was detected in a previous frame and the object tracker has now been able to recognize the tracked object as a specific identity, then the tracked object is included in the list of updated tracked objects.

8.2.4 DetectedObject

Common interface for all types of detected objects.

```
Source
/**
 * Common interface for all types of detected objects.
 */
public interface DetectedObject {

    /**
     * The type of the object
     *
     * @return
     */
    DetectedObjectType getObjectType();

    /**
     * The local ID of the object. Local IDs for detected objects are
        only unique with
     * respect to the single frame in which the object was detected.
     *
     * @return the local ID
     */
    Long getLocalId();

    /**
     * The bounding box of the object inside of the image in which it was
        detected.
     * Note that the bounding box is in normalized coordinates [0, 1] x
        [0, 1]
     *
     * @return
     */
    RectF getNormalizedBounds();

    /**
     * Returns the expansion factor by which the object bounds have been
       expanded
     * to cut out the object thumbnail. Note that the object thumbnail is
       usually
     * bigger than the area enclosed by the normalized bounds.
     * This is the factor by which  it was expanded.
     *
     * @return
     */
    float getThumbnailBoundsExpansionFactor();

    /**
     * How confident the object recognizer is that this is actually an
       object for really
     *
     * @return
     */
    float getConfidence();

    /**
     * How suitable is this object for object recognition?
     *
     * @return
     */
    double getCenterPoseQuality();

    /**
     * The yaw of the object measured as the object is moves left-right.
     *
     * @return -1 for left, 0 for center and +1 for right.
     */
    double getYaw();

    /**
     * The pitch of the object measured as the object moves up/down.
     *
     * @return  -1 for straight down, 0 for center and +1 for straight up.
     */
    double getPitch();

    /**
     * The roll of the object is measured as the object is tilted
       left-right.
     *
     * @return  -1 for left, 0 for center and +1 for the right.
     */
    double getRoll();
    /**
     * What is the sharpness score of the image that was used for
        detection?
     *
     * @return
     */
    double getImageSharpnessQuality();

    /**
     * What is the contrast quality of the image from which this object
        was taken?
     *
     * @return
     */
    double getImageContrastQuality();

    /**
     * The object thumbnail
     *
     * @return
     */
    Bitmap getThumbnail();

    /**
     * The scene image - the image of the scene from which the object
        thumbnail was extracted.
     *
     * @return
     */
    Bitmap getSceneThumbnail();

    /**
     * Data that may be passed to the object recognizer to speed up
        recognition
     *
     * @return
     */
    String getRecognizerHint();

    /**
     * Max ratio of clipping on either side: if the object is a
        potentially partial object
     *  which means that its bounding box extends beyond the video frame
        boundaries
     *
     * @return
     */
    float getClipRatio();

    /**
     * The bounding box before normalization.
     *
     * @return
     */
    RectF getPixelBounds();

    enum DetectedObjectType {
        badge,
        face,
        recognizedObject
    }
}
```

8.2.5 TrackedObject

A tracked object represents a single and unique instance of an object that the object tracker has been able to detect in the video stream which it is actively tracking. A tracked object has a type that indicates whether it is a badge(not yet supported) or the face of a person. A tracking object also has an axis-aligned bounding box which tells you where in the input video frame the tracked object can be found and what its size is.

TrackedObject provides the **getDetectedObject()** method whose return type is **DetectedObject** - a common interface for all types of detected objects and which is used to get the face info.

This can be referenced in the sample app's **GraphicOverlay** class, under the **drawFace()** method.

8.2.6 TrackedFace

A **TrackedFace** represents a human face. A **TrackedFace** may be linked to a person. If the object tracker is able to recognize the face as belonging to a specific person than the **TrackedFace** provides a reference to the corresponding **Person** object.

8.2.7 Person

A **Person** object provides information about a person that has been registered with the ArgusKit face recognition service. Each person has a unique identifier. You may assign a name and a set of tags to a person with the help of a **TrackedFaceChange** object and the **ObjectTracker.apply()** object tracker function.

8.2.8 TrackedFaceChange

A person change object stores attributes that should be applied to the person record on file in the ArgusKit face recognition service. This object allows you to change a person's name, tags, age, or gender information.

8.2.9 VideoFrame

A **VideoFrame** object encapsulates a single decoded video frame which should be passed to an object tracker instance.

8.3 Event Reporting

This section describes the SDK objects which you use to generate events from object tracker results and to post them to the event server in the SAFR Cloud.

8.3.1 ObjectEventDataStore

This object acts as a local store for events. It is needed by the **PeopleIndexer** object which is the main interface to the event reporting system.

8.3.2 ObjectEventLog

The object event log records object tracker results. You should invoke the **update()** method with the current object tracker result every time the object tracker invokes your **didTrack()** event handler.

8.3.3 PeopleIndexer

The **PeopleIndexer** object ties together the object event store and the object event log. In essence, it coordinates the event reporting mechanism. Pass your configuration object when you create an instance of the people indexer.

The **PeopleIndexer** provides hooks/callbacks which are invoked when an event changes its state or when it's done processing events.

```
interface PeopleIndexerDelegate {
    /**
     * Called when the event is started.
     */
    fun peopleIndexerDidStartEvent(peopleIndexer: PeopleIndexer?,
       personEvent: PersonEvent?)

    /**
     * Called when the event is updated.
     */
    fun peopleIndexerDidUpdateEvent(
        peopleIndexer: PeopleIndexer?,
        personEvent: PersonEvent?,
        updatedProperties: PersonEventUpdatableProperties?
    )

    /**
     * Called when the event had ended.
     */
    fun peopleIndexerDidEndEvent(peopleIndexer: PeopleIndexer?,
       personEvent: PersonEvent?)

    /**
     * Called when there are no ongoing events
     */
    fun peopleIndexerDidEndProcessingEvents(peopleIndexer: PeopleIndexer?)
}
```

8.3.4 PeopleIndexer.Configuration

The **PeopleIndexer.Configuration** object stores all relevant configuration information for the event reporting system. Create an instance of this object to get a default configuration and then set the cloud account and environment information to enable reporting to the SAFR Cloud event server.

8.4 Analyzing Images

This section describes the image analyzer object which is used to find faces in an image.

8.4.1 ImageAnalyzer

A utility class that does a single image analysis. It processes an image in order to find faces. Object detection and recognition are executed in real time.

8.4.2 ImageAnalysisResult

An **ImageAnalysisResult** object contains a snapshot of what was detected in an image. The image analyzer result contains a list of objects that have appeared in the current image.

8.4.3 Action Recognition

8.4.3.1 LivenessActionRecognizer Used for RGB liveness detection. Feed it with **ObjectTracker**'s **TrackingResult**. The delegate's **didRecognizeObjectAction** callback is invoked when the liveness state for a given **TrackedObject** is concluded.

8.4.3.2 SmileActionRecognizer Used for smile detection. Feed it with **ObjectTracker**'s **TrackingResult**. The delegate's **didRecognizeObjectAction** callback is invoked when the **detectedSmile** state for a given **TrackedObject** is concluded.

8.5 Settings and ModeSettings Configuration Options

Generally, **Settings** should be configured before initializing/creating other SDK objects. The **Settings** object provides an option to register a callback, which is invoked when any of the settings are changed.

To prevent multiple callback invocations, settings should be changed in bulk mode:

```
settings
// Begin bulk change
settings.beginBulkChange();
// update configuration

settings.setUserIdentifier("user");
settings.setUserPassword("password");
settings.setUserDirectory("directory");
ModeSettings modeSettings = settings.getModeSettings();

modeSettings.setFaceRecognizerDetectIdentity(true);
modeSettings.setFaceRecognizerDetectGender(true);
modeSettings.setFaceRecognizerDetectAge(false);
modeSettings.setFaceRecognizerDetectSentiment(true);
modeSettings.setFaceRecognizerDetectOcclusion(false);

// End bulk change
settings.endBulkChange();
```

Once done, the callback will be invoked with the list of changed keys:

```
@Override
public void onSettingsChange(@NonNull HashSet<String> hashSet) {

    // i.e. recreate SDK objects

    // Or restart activity
    new Handler().post(() -> getActivity().recreate());
}
```

9 Android SDK Configuration Options

9.1 Recognizer

```
// Cloud user password
Settings.userPassword() : String

// Cloud user identifier
Settings.userIdentifier() : String

// Cloud user directory
Settings.userDirectory() : String

// Cloud user directory for similar mode
Settings.userSimilarDirectory() : String

// User source to be reported to the cloud
Settings.userSource() : String

// User site to be reported to the cloud
Settings.userSite() : String

// Cloud environment
Settings.cloudEnvironment() : CloudEnvironment

// Whether to enable detection of an identity, which matches against the
   existing database of people (identities).
faceRecognizerDetectIdentity(): Boolean

// Whether to enable the detection of gender information.
faceRecognizerDetectGender(): Boolean

// Whether to enable the detection of age information.
faceRecognizerDetectAge(): Boolean

// Whether to enable occlusion detection during recognition.
faceRecognizerDetectOcclusion(): Boolean

// Whether to enable mask detection during recognition.
faceRecognizerDetectMask(): Boolean

// Whether to enable the detection of sentiment information.
faceRecognizerDetectSentiment(): Boolean

// The minimum size of faces to detect. This value is applied after
   searching the image.
faceRecognizerMinimumFaceSize(): Int

// The minimum resolution that a recognition candidate image must have in
   order to allow the insertion of the candidate image into the Cloud
   database.
faceRecognizerIdentificationMinimumFaceSize(): Int

// The amount of time that the no-smile should last
faceRecognizerNoSmileActionDuration(): Double
```

```
// The amount of time that the smile should last
faceRecognizerSmileActionDuration(): Double

// When smile action recognition is enabled, this property boosts level
   of face difference tolerated when confirming identity via a face in
   smiling expression.
// Smile action recognition must first recognize identity at level of
   difference defined by recognizer.identity-recognition-threshold +
   recognizer.identity-proximity-threshold-allowance (e.g.  0.54 + 0.0 =
   0.54 which is a confident match).
// Face must be non-smiling (serious) at this beginning stage for the
   specified recognizer.smile-pre-delay duration.
// After initial repeated recognition of non-smiling face for the
   specified amount of time, recognition will be repeated once face
   transitions into smiling expression.
// When in smiling expression, identity recognition will be accepted at
   the level of face difference recognizer.identity-recognition-threshold
   + recognizer.identity-proximity-threshold-allowance +
   recognizer.smile-identity-threshold-boost (e.g. 0.54 + 0.0 + 0.13 =
   0.67 which is a close match but not certain).
// Thus, when in smiling expression, greater level of face difference is
   tolerated.  If this value is set to 0.0, smiling face needs to be
   recognized at the same level of strictness as the initial serious
   expression face in order for simile action (mapped to smileToActivate
   actionId event) to be activated.
faceRecognizerSmileActionIdentityRecognitionThresholdBoost(): Double

// The identity recognition threshold proximity allowance defines level
   of difference between faces in addition to what is specified in
   recognizer.identity-recognition-threshold for which matches will be
   reported.
// Possible value range is from 0.0 to 4.0.
// Is set to 0.0, no additional face difference beyond what is specified
   in recognizer.identity-recognition-threshold will be reported on.
// This means that only 100% (and higher confidence matches) will be
   reported.
// When set to value greater than 0, system will report matches less than
   100% up to the allowance provided.
// For example, if recognizer.identity-recognition-threshold = 0.54 and
   recognizer.identity-proximity-threshold-allowance = 0.38,  system will
   reports matches with face ranging in level of difference from 0 to
   0.92  (0.54 + 0.38) .  Thus SAFR will report on all faces that have
   some similarity (see below table on how to interpret face difference
   level).  100% similarity score will be given to faces with difference
   level of 0.54.
//
// Guidelines on interpreting face difference level are as follows:
// 0.00 - identical face image match
// <0.00  0.30]   -  extremely confident match  (e.g. high security area
   entry under controlled environmental conditions)
// <0.30  0.45]   -  very confident match   (e.g. unlock door for
   un-monitored facility)
// <0.45  0.54]   -  confident match  (e.g. unlock door for monitored
   facility)
// <0.54  0.67]   -  close match but not certain
// <0.67  0.84]   -  possible match with low confidence
// <0.84  0.92]   -  similar face with no confidence of match
// <0.92  4.00]   -  face without significant similarity
//
// Similarity score (% match) is computed based on difference level as
   follows:
// similarityScore = (2 - SQRT(faceDifferenceLevel)) / (2 -
   SQRT(recognizer.identity-recognition-threshold))
faceRecognizerIdentityRecognitionThresholdProximityAllowance(): Double

// Enables the smile threshold values
faceRecognizerSmileActionSmileTransitionThresholdsEnabled(): Boolean

// The threshold in which there is no smile
faceRecognizerSmileActionNoSmileThreshold(): Double

// The threshold in which there is a smile
faceRecognizerSmileActionSmileThreshold(): Double

// The identity recognition threshold represents maximum level of
   difference (distance) between two faces for them to be considered a
   certain identity match (100% match).
// Possible range of values is between 0.0 and 4.0.
// Value of 0.0 means that for two faces to match at 100%, the images
   representing them  would need to be identical in every pixel.
// Value of 4.0 means that any two images are considered a certain match.
    Default value of 0.54 is calibrated to reflect match certainty
   appropriate for secure access use-cases.
//
// Guidelines on the meaning of the value of this threshold are as
   follows:
// 0.00 - identical face image match
// <0.00  0.30]   -  extremely confident match  (e.g. high security area
   entry under controlled environmental conditions)
// <0.30  0.45]   -  very confident match   (e.g. unlock door for
   un-monitored facility)
// <0.45  0.54]   -  confident match  (e.g. unlock door for monitored
   facility)
// <0.54  0.67]   -  close match but not certain
// <0.67  0.84]   -  possible match with low confidence
// <0.84  0.92]   -  similar face with no confidence of match
// <0.92  4.00]   -  face without significant similarity
faceRecognizerIdentityRecognitionThresholdCamera(): Float

// The identity recognition threshold for similar mode.
faceRecognizerIdentityRecognitionThresholdSimilar(): Float

// Valid values are in the range of 0.0 - 1.0. This is the maximum
   occlusion value that is allowed when inserting new recognition
   candidate images into the Cloud database. If the face is occluded with
   a value greater than this then the face will not be added, but if it
   is less than or equal to this value then it will be added
```

```
faceRecognizerMaxOcclusion(): Double

// Mask detection confidence at which system is to assume presence of the
   mask. Possible value range is between 0 and 1.0 . By increasing the
   value, number of false positive mask detection will be decreased at
   the expense of lower true positive mask detections. By decreasing this
   value, number of true positive mask detections will be increased at
   the expense of higher false positive mask detections.
faceRecognizerMaskThreshold(): Double

// Mask model "sensitive", "precise", "normal"
faceRecognizerMaskModel(): String

// The maximum clip ratio on either side the recognition candidate might
   have
faceRecognizerClippingTolerance(): Double

// The maximum clip ratio on either side the insertion candidate might
   have
faceRecognizerIdentificationClippingTolerance(): Double

// This offset adjusts recognizer.identity-recognition-threshold to be
   applied to masked faces.
// Thus, definition of 100% identity match can be customized for masked
   faces.  By increasing this offset, 100% identity match definition will
   be loosened thus allowing more true positives to be included under
   100% match criteria at the expense of more false positives.  By
   decreasing this offset, 100% match for masked faces will be made more
   strict thus reducing the number of false positive at the expense of
   fewer true positives.
// For example, if recognizer.identity-recognition-threshold = 0.54 and
   recognizer.identity-masked-threshold-offset = -0.09, 100% identity
   match for faces without masks will be declared at face level
   difference of 0.54 (confident match), while 100% identity match for
   masked faces will be declared at face level difference of 0.45 (a very
   confident match).   0.45 face level difference is arrived at by adding
   recognizer.identity-masked-threshold-offset to
   recognizer.identity-recognition-threshold.  For above example, 0.54 -
   0.09 = 0.45 .
// Note that for this offset to be applied, mask must be detected on a
   face.  That is only possible if mask detection is enabled via
   recognizer.detectMask = true property.
faceRecognizerIdentityMaskedThresholdOffset(): Double

// The maximum number of similar people to fetch
faceRecognizerSimilarLimit(): Int

// The minimum CPQ that a recognition candidate must have in order to
   allow the insertion of the candidate image into the Cloud database.
faceRecognizerIdentificationMinimumCenterPoseQuality(): Double

// The minimum FSQ that a recognition candidate must have in order to
   allow the insertion of the candidate image into the Cloud database
faceRecognizerIdentificationMinimumFaceSharpnessQuality(): Double
```

```
// The minimum FCQ that a recognition candidate must have in order to
   allow the insertion of the candidate image into the Cloud database
faceRecognizerIdentificationMinimumFaceContrastQuality(): Double

// The minimum center pose quality that a face image must have before we
   try to recognize the face.
faceRecognizerMinimumCenterPoseQuality(): Double

// The minimum face sharpness quality that a face image must have before
   we try to recognize the face.
faceRecognizerMinimumFaceSharpnessQuality(): Double

// The minimum face contrast quality that a face image must have before
   we try to recognize the face.
faceRecognizerMinimumFaceContrastQuality(): Double

// The minimum resolution a recognition candidate must have in order to
   allow merging
faceRecognizerMergingMinimumFaceSize(): Int

// The minimum CPQ that a recognition candidate must have in order to
   allow merging
faceRecognizerMergingMinimumCenterPoseQuality(): Double

// The minimum FSQ that a recognition candidate must have in order to
   allow merging
faceRecognizerMergingMinimumFaceSharpnessQuality(): Double

// The minimum FCQ that a recognition candidate must have in order to
   allow merging
faceRecognizerMergingMinimumFaceContrastQuality(): Double
```

9.2 Detector and ObjectTracker

```
// High sensitivity detector input size [0 - Extra small; 1 - Small; 2 -
   Normal; 3 - Large].
// The higher the value, the better the accuracy but at the cost of speed.
Settings.retinaInputSize(): Int

// The initial (1 of 3) face candidate threshold that is used during face
   detection when using "detector.detect-faces-service" set to normal.
// This parameter is for highly advanced usage of "normal" face detection
   service.
// It should be used only if optimizing face detector for performance on
   specialized type of data.
// This parameter has no effect when "detector.detect-faces-service" is
   set to other than "normal"  (e.g. "high-sensitivity").
initialCandidateThreshold(): Float

// The middle (2 of 3) face candidate threshold that is used during face
   detection when using "detector.detect-faces-service" set to normal.
// This parameter is for highly advanced usage of "normal" face detection
   service.
```

```
// It should be used only if optimizing face detector for performance on
   specialized type of data.
// This parameter has no effect when "detector.detect-faces-service" is
   set to other than "normal"  (e.g. "high-sensitivity").
middleCandidateThreshold(): Float

// The final (3 of 3) face candidate threshold that is used during face
   detection when using "detector.detect-faces-service" set to normal.
// This parameter is for highly advanced usage of "normal" face detection
   service.
// It should be used only if optimizing face detector for performance on
   specialized type of data.
// This parameter has no effect when "detector.detect-faces-service" is
   set to other than "normal"  (e.g. "high-sensitivity").
finalCandidateThreshold(): Float

// Filter candidates after nms
retinaFilterThreshold(): Float

// Whether to enable face detection
faceDetectorEnabled(): Boolean

// Whether to generate and send recogniser hint for detected faces
faceDetectorSendRecognizerHint(): Boolean

// Selected face detector model 1 - standard; 2 - high definition
faceDetectorModel(): Int

// The expansion factor for detected face thumbnails
faceDetectorObjectThumbnailSizeExpansionFactor(): Float

// The minimum number of consecutive recognition attempts that we must
   run and produce the same person identity before we lock onto this
   identity and learn it
// (Insert it into the server database).
objectTrackerMinimumNumberOfIdenticalRecognitionsToLearn(): Int

// The time in-between reconfirmation attempts.
// We reconfirm the identity of a tracked face from time to time after it
   has been recognized as a person X.
objectTrackerReconfirmationTimeInterval(): Microseconds

// The object detector is advised to search for objects of at least this
   size.
// This value is applied while searching the image.
faceDetectorMinimumSearchedFaceSize(): Int

// The minimum size of faces to accept from the detector.
// Only faces with at least this size are eligible for recognition.
faceDetectorMinimumRequiredFaceSize(): Int

// Determines for how many frames more we continue to keep a tracked face
   around after we have failed to detect it in the most recent frame.
// This makes the tracker resilient against intermittent loss of face.
```

```
objectTrackerMaximumLingerFrames(): Int

// The minimum number of consecutive recognition attempts that we must
   run and produce the same person identity before we lock onto this
   identity.
objectTrackerMinimumNumberOfIdenticalRecognitionsToLock(): Int

// Whether to enable correlation of tracked faces and detected faces by
   comparing the change in area.
objectTrackerFaceSizeCorrelation(): Boolean

// Whether identity of tracked face is removed every time recognition
   fails on high quality (merge level) face
objectTrackerRemoveIdentityOnFailedRecognition(): Boolean
```

9.3 Event Reporting

```
// Whether to enable event reporter
eventsReportEnabled(): Boolean

// Wheteher report events for recognized people only.
// If this is true events are only reported for people who are recognized
   otherwise events are reported for all people detected.
eventsReportRecognizedPersonEventsOnlyCamera(): Boolean

// If this is true then stranger person events are allowed to be
   generated,
// otherwise the stranger person events will be converted to
   unidentified/unrecognized person events that are generated instead.
eventsReportStrangerPersonEvents(): Boolean

// If this is valid then this will be used for the lower bound of the
   stranger age range.
// This means the person will be classified as a stranger only if their
   age is above this value
// (or their age is not available/wasn't detected) otherwise they will be
   classified as an unidentified person.
eventsStrangerMinAge(): Int

// If this is valid then this will be used for the upper bound of the
   stranger age range.
// This means the person will be classified as a stranger only if their
   age is below this value
// (or their age is not available/wasn't detected) otherwise they will be
   classified as an unidentified person.
eventsStrangerMaxAge(): Int

// Reports events for speculated people.
// Speculated people are faces that aren't a 100% match but are close.
eventsReportSpeculatedPersonEvents(): Boolean

// Enables the inclusion of face thumbnails in event reports.
eventsReportSaveFaceImage(): Boolean
```

```
// Enables the inclusion of scene images in event reports.
eventsReportSaveSceneImage(): Boolean

// Delay the event reporting to the server by this amount in seconds.
eventsReportDelay(): Double

// The minimum allowed recognized person event duration in seconds.
// Events shorter than this value will not be reported.
eventsReportMinRecognizedPersonEventDuration(): Double

//The minimum allowed unrecognized person event duration in seconds.
// Events shorter than this value will not be reported.
eventsReportMinUnrecognizedPersonEventDuration(): Double

// The minimum allowed stranger event duration in seconds.
// Events shorter than this value will not be reported.
eventsReportMinUnrecognizedPersonEventDuration(): Double

// Whether the event reporter should monitor for event replies after
   adding the event
eventsEventReplyEnabled(): Boolean

// Whether the event reporter should update the face/scene/etc images for
   the event on the image server if the images are marked as being
   changed during an update.
eventsReportUpdateImages(): Boolean

// Filter for reported event types PersonEventType.person;
   PersonEventType.action.
eventsReportEventTypes(): Set<PersonEventType>?
```

9.4 RGB Liveness Detection

```
// Whether RGB liveness detector is enabled
livenessDetectorEnabled(): Boolean

// The minimum CPQ value required for liveness model to be evaluated
livenessMinCPQ(): Float

// The minimum sharpness value required for liveness model to be evaluated
livenessMinSharpness(): Float

// The minimum contrast value required for liveness model to be evaluated
livenessMinContrast(): Float

// The minimum face required for liveness model to be evaluated
livenessMinFaceSize(): Int

// The minimum Face context size (normalized %) for liveness detection
livenessMinFaceContextSize(): Float

// Liveness detection threshold - face with liveness value >= this
   threshold is considered alive
livenessDetectionThreshold(): Float
```

```
// Fake detection threshold - face with liveness value < this threshold
   is considered fake
livenessFakeDetectionThreshold(): Float

// Initial threshold used to short-circuit stage 2 liveness
livenessInitialDetectionThreshold(): Float

// Liveness detector scheme [ 0 - Texture Unimodal; 1 - Context Unimodal;
   2 - Strict Multimodal; 3 - Normal Multimodal; 4 -Tolerant Multimodal]
livenessDetectorScheme(): Int

// Number of frames to evaulate live detections
livenessEvaluationFrameCount(): Int

// Number of frames to evaulate fake detections
livenessFakeEvaluationFrameCount(): Int

// Minimum confirmations threshold to conclude liveness
livenessConfirmationsThreshold(): Float
```

10 Offline vs. Online Modes

The Offline and Online modes are configurable via the **Settings** object:

```
public void setIsOfflineMode(boolean flag);
public boolean isOfflineMode();
```

- In Offline mode, the SAFR SDK performs face detection, face tracking, and face-quality assessment; no network communication is required.
- In Online mode, the SAFR SDK also performs face recognition for which the network communication and a valid SAFR account are required.

11 FAQ

11.1 How Can I Set Up and Preview Camera Feeds?

The Android SDK offers two camera implementations: **VisionCameraSource** and **CameraXSource**. (We recommend that you use **CameraXSource**.) A preview (**CameraXSourcePreview** or **VisionCameraSourcePreview**) should be added to the view hierarchy and passed into the constructor.

Relevant sample code can be found in the **createCameraSource** method within the sample app.

11.2 What Do I Do If a Device Is Experiencing Low Performance? (Few Detections Per Second)

This can happen on lower end devices, in which case you can try the following:

1. Reduce Camera frame resolution. Loop through the returned camera profiles in **createCameraSource** and select a lower resolution.

   ```
   final CameraProfile cp = cameraProfiles.get(facing).get(0); // Select
       another instead of the first one
   ```

2. If you're using a High Sensitivity detector, try reducing the detector input size.

   ```
   settings.setRetinaInputSize(...) // 0 - Extra small; 1 - Small; 2 -
       Normal; 3 - Large
   ```

3. Switch to the Standard detector. (Note that this will adversely impact the detector's accuracy.)

   ```
   //faceDetectorServiceConfig.detectorModel = FDRuntimeOptions.FD_STANDARD;
       // Uncomment to enable
   ```

11.3 How Can I Extract and Save Detected Faces?

DetectedObject is used to obtain a face bitmap; via the **DetectedObject.getThumbnail()** method in particular. The method returns a downscaled bitmap where the size depends on face's aspect ratio. (The smaller side will be 240px.)

If a different size is needed, the **DetectedObject** also includes the pixel bounds, required to extract a face from the original frame.

The sample code can be found in **ShareManger.share()** where face thumbnail is saved to the local cache storage and shared via Android share intent.

11.4 How Can I Use RGB Liveness Detection?

1. Initialize **livenessActionRecogniser**.

   ```
   livenessActionRecognizer = new LivenessActionRecognizer(settings);
   livenessActionRecognizer.setDelegate(this);
   ```

2. Pass tracking results to the liveness recognizer.

   ```
   if (livenessActionRecognizer != null) {
       livenessActionRecognizer.update(trackingResult);
   }
   ```

3. Read liveness info once received.

```
final Boolean isLivenessConfirmed = trackedObject.isLivenessConfirmed();
    if (isLivenessConfirmed) {
    //trackedObject.getLivenessConfidence()
    }
}
```

4. Pass the liveness object action to the object event log.

```
@Override
public void didRecognizeObjectAction(ObjectActionRecognizer
   objectActionRecognizer, ObjectAction objectAction, TrackedObject
   trackedObject) {
    if (objectActionRecognizer == livenessActionRecognizer) {
        DispatchQueue.Main.async(() ->
           overlay.onFaceLivenessConfirmed(trackedObject));

        // Propagate the objectAction to the objectEventLog in order to
           report it
        if (objectEventLog != null) {
            objectEventLog.update(trackedObject, objectAction);
        }
    }
}
```

11.5 How Can I Play rtsp:// Streams?

Use **AglVideoView** to play rtsp:// streams.

The sample code can be found in *VidiaFragment.java.* Search for **aglView** references.

11.6 How Can I Configure AglVideoView to Use HW Decoder?

Set the decoder type option to *AglOptions.DECODER_MEDIACODEC*.

The sample code can be found in *VidiaFragment.java.* Search for **aglView** references.

11.7 How Can I Enable Event Reporting?

Initialize **ObjectEventLog** and **PeopleIndexer** and pass **TrackingResult** in **ObjectEventLog.update()**.

The sample code can be found in *VidiaFragment.java.* Search for **objectEventlog** references.

11.8 How Can I Process rtsp:// Streams in the Background?

The demo app provides an example service which plays rtsp:// streams and feeds frames into the **ObjectTracker** for face detection. A notification is posted when a person from the stream is recognised.

Usually, the service should be started/stopped from Activity's **onPause()** and **onResume()** callbacks.

This isn't SDK-related but there is a sample code to schedule start/stop of the service, located in *VidiaFragment.java.* Search for **CameraProcessorService** references.