**TECHSIGN RKYC**

**Android**

* Supported minimum sdk version is Android 4.0.3 Ice Cream Sandwich (API 15) which can run on approximately 100% of devices according to Android Studio.

**Integration**

We have two libraries namely card.io and rkyc-android which are packaged as aar files. You can integrate these libraries by,

1- Adding a *“libs”* folder under the *“app”* folder.

2- Adding below lines to *“app/build.gradle”* under *“android”* tag

*repositories {*

*flatDir {*

*dirs 'libs'*

*}*

*}*

3- Adding below lines to *“app/build.gradle”* under *“dependencies”* tag

*implementation(name: 'card.io-release', ext: 'aar')*

*implementation(name: 'rkyc-android-lib', ext: 'aar')*

*//nfc*

*implementation 'com.madgag.spongycastle:core:1.58.0.0'*

*implementation 'com.madgag.spongycastle:bcpkix-jdk15on:1.58.0.0'*

*implementation 'org.jmrtd:jmrtd:0.5.13'*

*implementation 'edu.ucar:jj2000:5.2'*

*implementation 'com.github.mhshams:jnbis:1.1.0'*

*//server call*

*implementation 'com.github.scribejava:scribejava-core:4.0.0'*

*implementation group: 'commons-io', name: 'commons-io', version: '2.4'*

*implementation 'net.sf.scuba:scuba-sc-android:0.0.9'server call*

4- Adding below lines to *“app/src/main/AndroidManifest.xml”* under *“application”* tag

*<activity*

*android:name="io.card.payment.TechsignScanActivity"*

*android:configChanges="keyboardHidden|orientation"*

*android:screenOrientation="portrait" />*

**Proguard**

-keep class net.sf.scuba.\*\* { \*; }

-keep class com.techsign.\*\* { \*; }

-keep class org.bytedeco.\*\* { \*; }

-keep class org.bouncycastle.\*\* { \*; }

-keep class org.spongycastle.\*\* { \*; }

**API**

Test API Root URL:

<http://169.62.55.86:8098/>

RKYC Server API is based on REST. API calls used for operations are listed in the following sections, together with the corresponding operation. For every API call, the type of the operation (POST/GET), the input and output models are listed, which are typically JSON unless mentioned otherwise.

**Usage**

Our service calls are on ServerCall class, the service calls run async and take a TechSignServiceListener which calls onSuccess method when it is successful and calls onFailure method when there is an error.

**1- Login**



Login service can be used by calling ServerCall.login() function which requires a FaceAuthorizationModel and a TechSignServiceListener<VerifyFaceReturnModel>.

LoginModel consists of three fields,

* String “id” => citizen id
* String “video” => base64 of recorded selfie video[1]
* int “rotate” => rotation of recorded video (0,90,180 or 270)

[1]. The recorded video should be around 5 seconds and 480p.

If the service call is successful, onSuccess method called with a VerifyFaceReturnModel which consist of,

* String “id” => citizen id
* String “name” => user’s name
* String “surname” => user’s surname
* String “birthDate” => user’s birth date[2]
* String “expireDate” => user’s expire date[2]
* String “picture” => base64 of user’s picture on identity card which he/she used for registration
* String “gesture” => which gesture is used for liveness check[3]
* ControlEntryModel[] “controlResults” => ControlEntryModel consists of,
* String “entryType” => ControlEntryTypes enum[4]
* Float “entryScore” => the result of the control between 0 and 1.
* Boolean “entryInterpretation” => true if the control is successful
* String “controlId” => unique identifier of the control

[2]. dates are in YYMMDD format.

[3]. we have two gesture types,

* “MOUTH” => user should open and close his/her mouth at least two types in 5 seconds.
* “EYE” => user should open and close his/her eyes at least two types in 5 seconds.

[4].public enum ControlEntryTypes {

PhotoCheatVerification("PHOTOCHEAT"),

HologramVerification("HOLOGRAM"),

HologramFaceVerification("HOLOGRAMFACE"),

FaceLivenessVerification("FACELIVENESS"),

FaceVerification("FACE"),

HiddenPhotoVerification("HIDDENPHOTO"),

SignaturePhotoVerification("SIGNATUREPHOTO"),

GuillocheVerification("GUILLOCHEVERIFICATION"),

RainbowVerification("RAINBOWVERIFICATION"),

OCRNFCSimilarity("OCRNFCSIMILARITY"),

OCRMRZSimilarity("OCRMRZSIMILARITY");

}

We are using the “EYE” gesture in login.

Note: For a successful login both *“*FaceVerification*”* and *“*FaceLivenessVerification*”* in “controlResults” should be passed.

**API Call:**

**Server Suffix:** /face/authorize

**Type: POST**

**Input:**

{

id:String //ID Number for login, should be registered before

video: String //captured liveness video, in Base64 format

rotate: Int //clock-wise rotation to apply to the video before control, 0/90/180/270

}

**Output:**

VerifyFaceReturnModel, details listed above.

**2- Register**

****

Registration process starts with the card process and ends with the liveness process.

1. **Card Process**

To register, we capture your card image. You can use *“IDCardReader”* for this.

Initialize *“IDCardReader”.*

*idCardReader = new IDCardReader(this, new IDCardReader.IDCardReaderListener() {*

*@Override*

*public void onCanceled() {*

*//canceled*

*}*

*@Override*

*public void onProvided(OcrResultModel model) {*

*//provided*

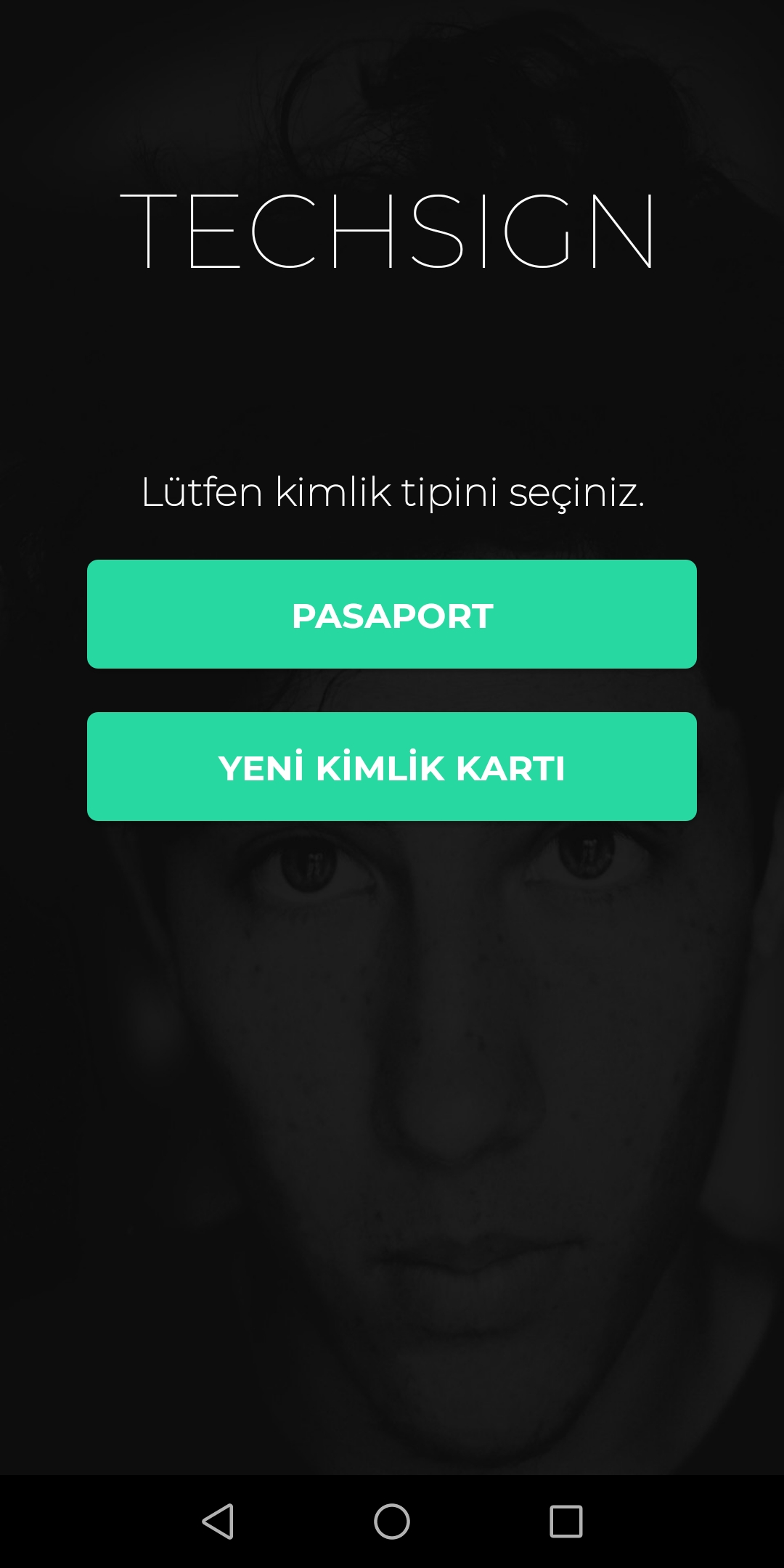
*}*

*})*

*;*

There are two methods for registering, OCR and NFC.

* **With NFC**

****

Start the *“IDCardReader”.*

*idCardReader.setIsMrzActive(true);*

*idCardReader.setCardType(card);*[4]

*idCardReader.setSkipOcr(true);*

*idCardReader.scan();*

[4]. The parameter *“card”* is an enum from *“model.Cardtype”.* Which consists of

* OLD\_ID
* NEW\_ID
* NEW\_ID\_BACK
* OLD\_DRIVER
* NEW\_DRIVER
* PASSPORT
* OLD\_ID\_BACK
* DRIVER\_BACK

For the NFC case, we should use PASSPORT or NEW\_ID\_BACK.

When the card image is captured the OCR process starts. When the OCR process is finished, we receive a callback to *“onProvided” with OcrResultModel.* Then we can get the detected card image and send it to the server for OCR.

*Bitmap bitmap = ocrResult.getCardPhoto();*

*base64 = bitmapToBase64(bitmap);*

*String cardTypeName = cardType == CardType.PASSPORT ? "PASSPORT" : "IDCARD";*

*ServerCall.verifyMrzOcr(new MrzOcrInputModel(base64, cardTypeName) ...*

**API Call:**

**Server Suffix:** /id/mrz-ocr

**Type: POST**

**Input:**

{

img:String //Base64 of captured card image

idType: String // “PASSPORT” for a passport and “IDCARD” for an id card

}

**Output:**

MrzOcrOutputModel, details listed below;

id:String => Citizen Number (TCKN)

name:String => Name

surname:String => Surname

birthdate:String => Birth Date in YYMMDD format

expiredate:String => Expire Date in YYMMDD format

docNumber:String => Document Number

****

We need to check if the device has NFC.

*NfcManager manager = (NfcManager) getSystemService(Context.NFC\_SERVICE);*

*NfcAdapter adapter = manager.getDefaultAdapter();*

*if(adapter == null){*

*// the device has not NFC*

*}else{*

*if(!adapter.isEnabled()){*

*//NFC is not enabled*

*//show android settings to make user enable NFC*

*Intent intent = new Intent(Settings.ACTION\_NFC\_SETTINGS);*

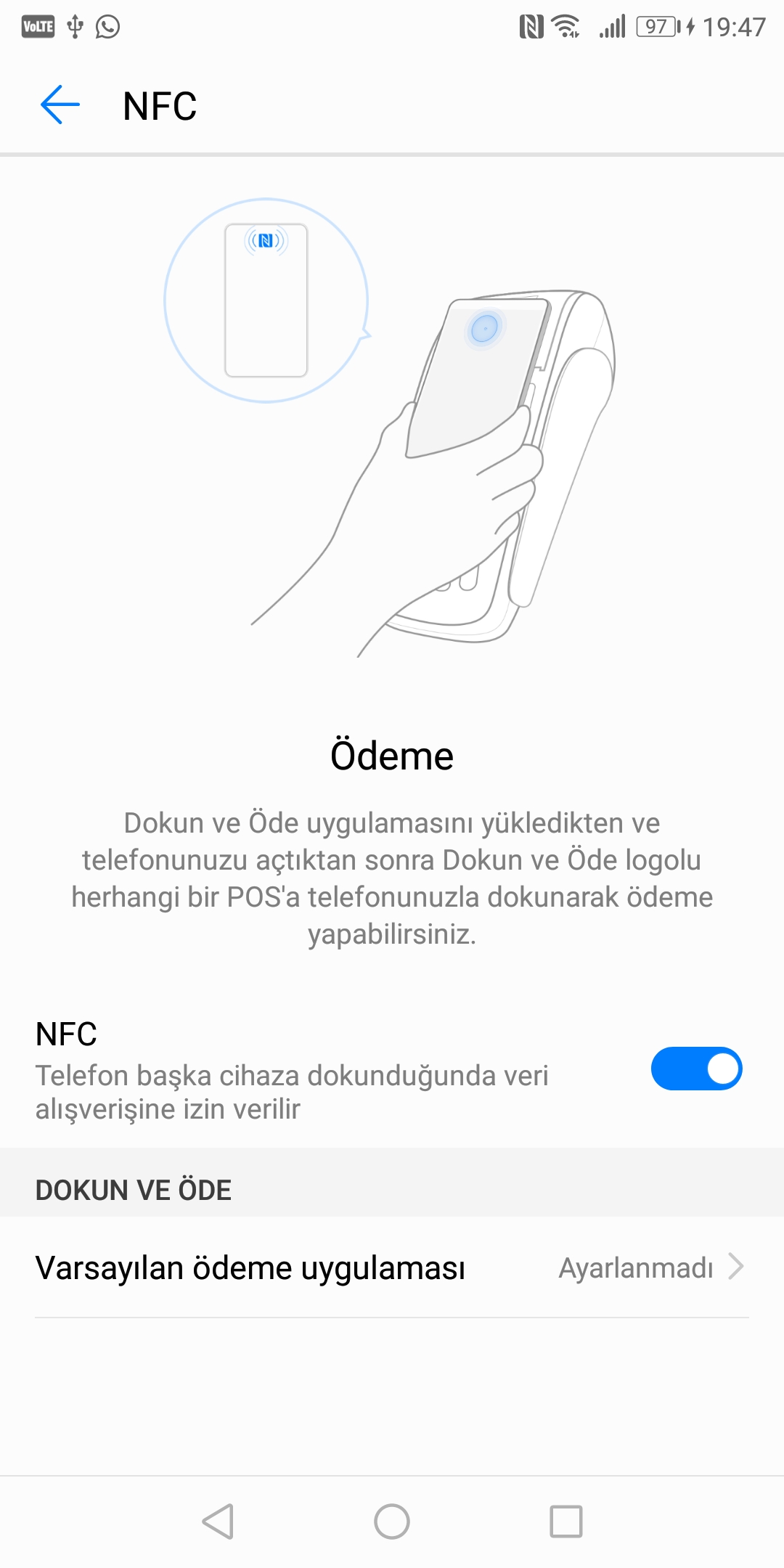
*startActivityForResult(intent,REQUEST\_NFC);*

*}else{*

*//NFC is enabled*

*}*

*}*

**

To check if user enabled the NFC,

*@Override*

*protected void onActivityResult(int requestCode, int resultCode, Intent data) {*

*super.onActivityResult(requestCode, resultCode, data);*

*if(requestCode == REQUEST\_NFC){*

*NfcManager manager = (NfcManager) getSystemService(Context.NFC\_SERVICE);*

*NfcAdapter adapter = manager.getDefaultAdapter();*

*if(adapter.isEnabled()){*

*// user enabled the NFC*

*}else{*

*// user has not enabled the NFC*

*}*

*}*

*}*

NFC verification requires document number, birth date and expire date. All of these are in *“OcrResult”* from *“onProvided”.* Note that the dates of *“OcrResult”* are in the form of DD.MM.YYYY. We need to convert them to YYMMDD form for the NFC verification. For NFC verification,

* We need to add this to the current activity;

*static {*

*Security.addProvider(new BouncyCastleProvider());*

*}*

* To start NFC we need to run these lines;

*NfcAdapter adapter = NfcAdapter.getDefaultAdapter(this);*

*if (adapter != null) {*

*Intent intent = new Intent(getApplicationContext(), this.getClass());*

*intent.setFlags(Intent.FLAG\_ACTIVITY\_SINGLE\_TOP);*

*PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, intent, PendingIntent.FLAG\_UPDATE\_CURRENT);*

*String[][] filter = new String[][]{new String[]{"android.nfc.tech.IsoDep"}};*

*adapter.enableForegroundDispatch(this, pendingIntent, null, filter);*

*}*

* To get NFC started callback, and start verification;

*@Override*

*public void onNewIntent(Intent intent) {*

*if (NfcAdapter.ACTION\_TECH\_DISCOVERED.equals(intent.getAction())) {*

*Tag tag = intent.getParcelableExtra(NfcAdapter.EXTRA\_TAG);*

*if (Arrays.asList(tag.getTechList()).contains("android.nfc.tech.IsoDep")) {*

*NFCReaderService nfcReaderService = new NFCReaderService(this);*

*nfcReaderService.readNFC(*

*IsoDep.get(tag),*

*cardType, // PASSPORT or NEW\_ID*

*docNumber,*

*birthDate,*

*expireDate,*

*new TechsignServiceListener<NFCDataModel>() {*

*@Override*

*public void onSuccess(NFCDataModel nfcDataModel) {*

*// nfc verification successful*

*}*

*@Override*

*public void onFailure(Exception e) {*

*if(e instanceof CardServiceException){*

*if(((CardServiceException) e).getSW() == ISO7816.SW\_SECURITY\_STATUS\_NOT\_SATISFIED){*

*//entered 3 fields are wrong (document no, expire date, birth date)*

*}else{*

*// other card exception*

*}*

*}else{*

*// other exception*

*}*

*}*

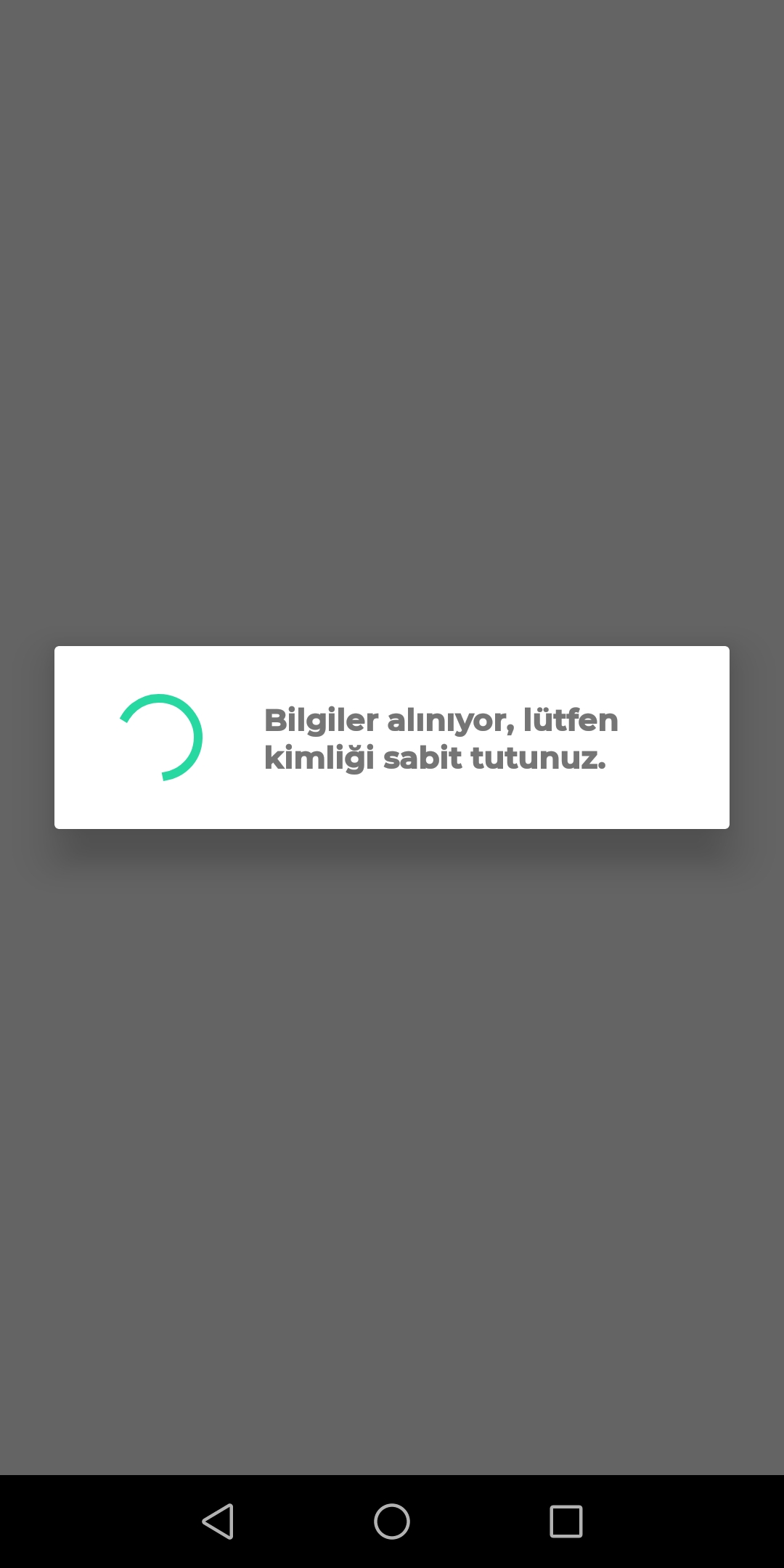
*});*

*}*

*}*

*}*

*}*

**

* To get notified for the process, you can set a *ServiceListener* to *NFCReaderService;*

*nfcReaderService.setServiceListener(new NFCReaderService.ServiceListener() {*

*@Override*

*public void process(NfcProcess nfcProcess) {*

*switch (nfcProcess){*

*case SECURITY:*

*//SECURITY CHECK FINISHED*

*Log.i(TAG, "process: security check finished");*

*break;*

*case ID:*

*//ID INFORMATION GATHERED*

*Log.i(TAG, "process: id information gathered");*

*break;*

*case CERTIFICATE:*

*//CERTIFICATE INFORMATIN GATHERED*

*Log.i(TAG, "process: certificate information gathered");*

*break;*

*case BIOMETRIC:*

*//BIOMETRIC INFORMATION GATHERED*

*Log.i(TAG, "process: biometric information gathered");*

*break;*

*}*

*}*

*});*

When the verification is successful we receive a *“NFCDataModel”* and we can convert it to “*ValidationInputModel*” by,

*IDValidationModel modelUnder = new IDValidationModel(nfcDataModel.getDg1(), nfcDataModel.getDg2(), nfcDataModel.getSod(), nfcDataModel.getDg11(), nfcDataModel.getDg12(), nfcDataModel.getDg14() ,nfcDataModel.getDg15())*

*idType = (cardType.equals(CardType.PASSPORT)) ? "PASSPORT" : "IDCARD";*

*ValidationInputModel modelValidate = new ValidationInputModel(modelUnder, idType,<TRANSACTION-ID>); // if the process started with ocr put the transactionId from CheckIdReturnModel else null*

Then using this model for ServerCall.validateAndSave(), we can receive *“SaveIdReturnModel”* which consists of,

* String *“transactionId”* => will be used in liveness process
* String *“gestureType”*  => which gesture should be used while recording the video “EYE” or “MOUTH”.
* IDValidationResultModel “validationResult” => Consists of 5 boolean;
* boolean dg1valid
* boolean dg2valid
* boolean sodvalid
* boolean docSigningValid
* boolean acceptable
* ControlEntryModel[] “controlResults” => ControlEntryModel consists of,
* String “entryType” => ControlEntryTypes enum[1]
* Float “entryScore” => the result of the control between 0 and 1.
* Boolean “entryInterpretation” => true if the control is successful
* String “controlId” => unique identifier of the control

[1].public enum ControlEntryTypes {

PhotoCheatVerification("PHOTOCHEAT"),

HologramVerification("HOLOGRAM"),

HologramFaceVerification("HOLOGRAMFACE"),

FaceLivenessVerification("FACELIVENESS"),

FaceVerification("FACE"),

HiddenPhotoVerification("HIDDENPHOTO"),

SignaturePhotoVerification("SIGNATUREPHOTO"),

GuillocheVerification("GUILLOCHEVERIFICATION"),

RainbowVerification("RAINBOWVERIFICATION"),

OCRNFCSimilarity("OCRNFCSIMILARITY"),

}

**API Call:**

**Server Suffix:** /id/validate-and-save/

**Type: POST**

**Input:**

{

validationInformation {

dg1:String // base64 of dg1 file in NFC chip

dg2:String // base64 of dg2 file in NFC chip

sod:String // base64 of sod file in NFC chip

dg11:String // base64 of dg11 file in NFC chip

dg12:String // base64 of dg12 file in NFC chip

dg14:String // base64 of dg14 file in NFC chip

dg15:String // base64 of dg15 file in NFC chip

}

idType: String // “PASSPORT” for a passport and “IDCARD” for an id card

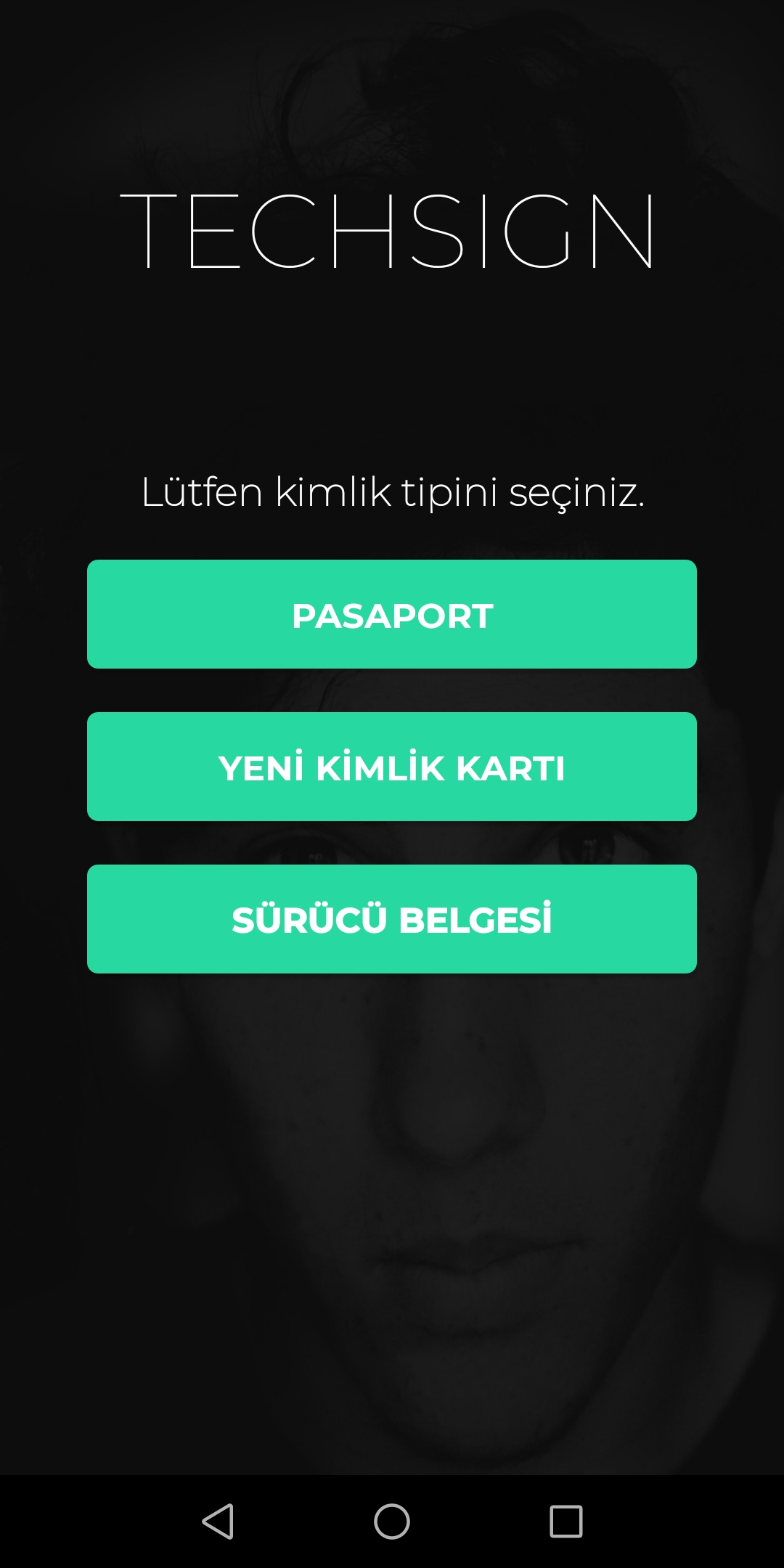
transactionId:String // if the transaction started with ocr process before nfc process the transactionId of the ocr process otherwise null

}

**Output:**

*SaveIdReturnModel*, details listed above.

* **With OCR**

****

We can start the OCR process with *“IDCardReader”.*

Start the *“IDCardReader”.*

*idCardReader.setIsMrzActive(false);*

*idCardReader.setCardType(card);*[1]

*idCardReader.setSkipOcr(true);*

*idCardReader.scan();*

[1]. The parameter *“card”* is an enum from *“model.Cardtype”.* Which consists of

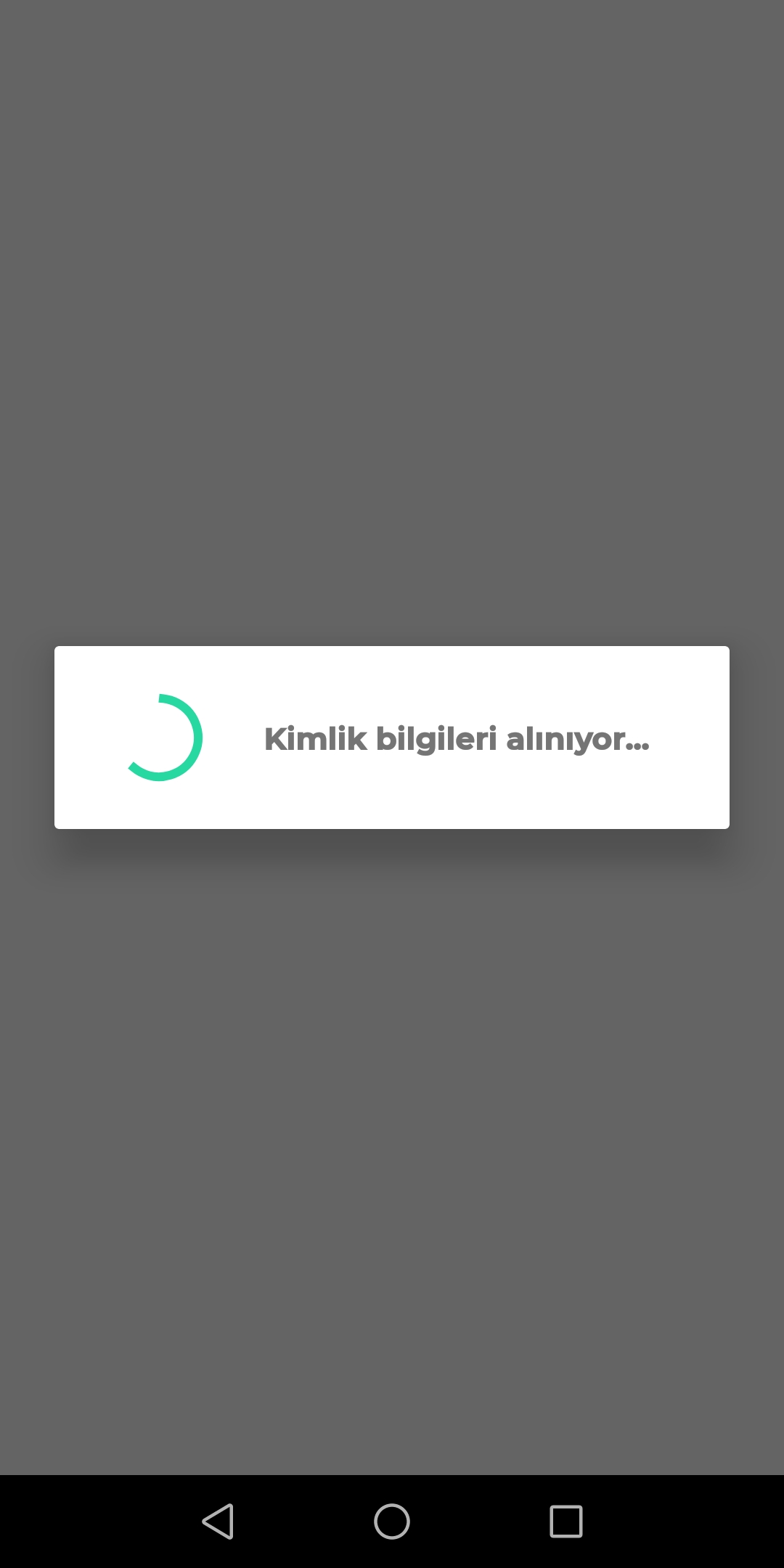
* OLD\_ID
* NEW\_ID
* NEW\_ID\_BACK
* OLD\_DRIVER
* NEW\_DRIVER
* PASSPORT
* OLD\_ID\_BACK
* DRIVER\_BACK

For the OCR case, we should use PASSPORT , NEW\_ID or NEW\_DRIVER.

When the card image is captured, we receive a callback to *“onProvided”.* We can get the card photo as Bitmap,

*Bitmap bitmap = ocrResult.getCardPhoto();*

Then according to the *“CardType”* we need to call ServerCall.verifyPassportPhoto(), ServerCall.verifyDriverPhoto() or ServerCall.verifyIDPhoto().

**

All of the calls require a CheckIDModel and a TechsignServiceListener<CheckIdReturnModel>.

CheckIDModel contains a String which is the base64 of the captured bitmap above.

CheckIdReturnModel consists of,

* String “transactionId” => used for hologram control and liveness check.
* String “gestureType” => which gesture will be used for liveness check.
* ControlEntryModel[] “controlResults” => ControlEntryModel consists of,
* String “entryType” => ControlEntryTypes enum[2]
* Float “entryScore” => the result of the control between 0 and 1.
* Boolean “entryInterpretation” => true if the control is successful
* String “controlId” => unique identifier of the control

.

[2].public enum ControlEntryTypes {

PhotoCheatVerification("PHOTOCHEAT"),

HologramVerification("HOLOGRAM"),

HologramFaceVerification("HOLOGRAMFACE"),

FaceLivenessVerification("FACELIVENESS"),

FaceVerification("FACE"),

HiddenPhotoVerification("HIDDENPHOTO"),

SignaturePhotoVerification("SIGNATUREPHOTO"),

GuillocheVerification("GUILLOCHEVERIFICATION"),

RainbowVerification("RAINBOWVERIFICATION"),

OCRNFCSimilarity("OCRNFCSIMILARITY"),

OCRMRZSimilarity("OCRMRZSIMILARITY");

}

*Note*: If the PhotoCheatVerification control in “controlResults” is successful we can continue to process. For the new id and new driver cards, the hologram process should be also completed before starting the liveness process.

**API Call:**

**Server Suffix:** /id/check/new-id

/id/check/new-driver

/id/check/passport

**Type: POST**

**Input:**

{

img:String //captured ID photo, in base64 format

transactionId:String // if the verification process started with nfc transaction id of nfc process else null

name:String // given name if not null OCR service calculates levenshtein distance and it is lower than %10 returns found name else returns given name, if no name present it should be null

surname:String // given surname if not null OCR service calculates levenshtein distance and it is lower than %10 returns found surname else returns given surname, if no surname present it should be null

expiredate:String // given expiredate if not null OCR service calculates levenshtein distance and it is lower than %10 returns found expiredate else returns given expiredate, if no expiredate present it should be null

birthdate:String // given birthdate if not null OCR service calculates levenshtein distance and it is lower than %10 returns found birthdate else returns given birthdate, if no birthdate present it should be null

}

**Output:**

CheckIdReturnModel, details listed above.

**Registration API Call with Only-OCR:**

This API Call collects OCR information for given ID Type, saves the transaction for registration, and returns the OCR result. Since checks are omitted, it is advised to use this API for only backend-to-backend integrations with logged users.

**Server Suffix:** /id/only-ocr

**Type: POST**

**Input:**

{

image:String //captured ID photo, in base64 format

imageType:String //Type of ID, "IDCARD", "PASSPORT", "DRIVERCARD", “OLDIDCARD"

}

**Output:**

* String “transactionId” => used for hologram control and liveness check.
* String “gestureType” => which gesture will be used for liveness check.
* String “id” => citizen id
* String “name” => user’s name
* String “surname” => user’s surname
* String “birthDate” => user’s birth date
* String “expireDate” => id’s expire date, if available
* String “serialNumber” => id’s serial number, if available

**HOLOGRAM**

ServerCall.detectHologram() function takes two parameters, a HologramDetectionInputModel and a TechsignServiceListener<HologramDetectionReturnModel>.

HologramDetectionInputModel consists of four fields,

* String “video” => base64 of recorded video[1]
* String “transactionId” => gathered from card process
* int “rotate” => rotation of recorded video (0,90,180 or 270)
* String “cardType” => “NEW-ID” for new id card, “NEW-DRIVER” for new drivers card.

HologramDetectionReturnModel consists of,

* ControlEntryModel[] “controlResults” => ControlEntryModel consists of,
* String “entryType” => ControlEntryTypes enum[2]
* Float “entryScore” => the result of the control between 0 and 1.
* Boolean “entryInterpretation” => true if the control is successful
* String “controlId” => unique identifier of the control

[1]. The recorded video should be around 5 seconds and 480p. The flash of the camera should be active after 1.5 second.

[2].public enum ControlEntryTypes {

PhotoCheatVerification("PHOTOCHEAT"),

HologramVerification("HOLOGRAM"),

HologramFaceVerification("HOLOGRAMFACE"),

FaceLivenessVerification("FACELIVENESS"),

FaceVerification("FACE"),

HiddenPhotoVerification("HIDDENPHOTO"),

SignaturePhotoVerification("SIGNATUREPHOTO"),

GuillocheVerification("GUILLOCHEVERIFICATION"),

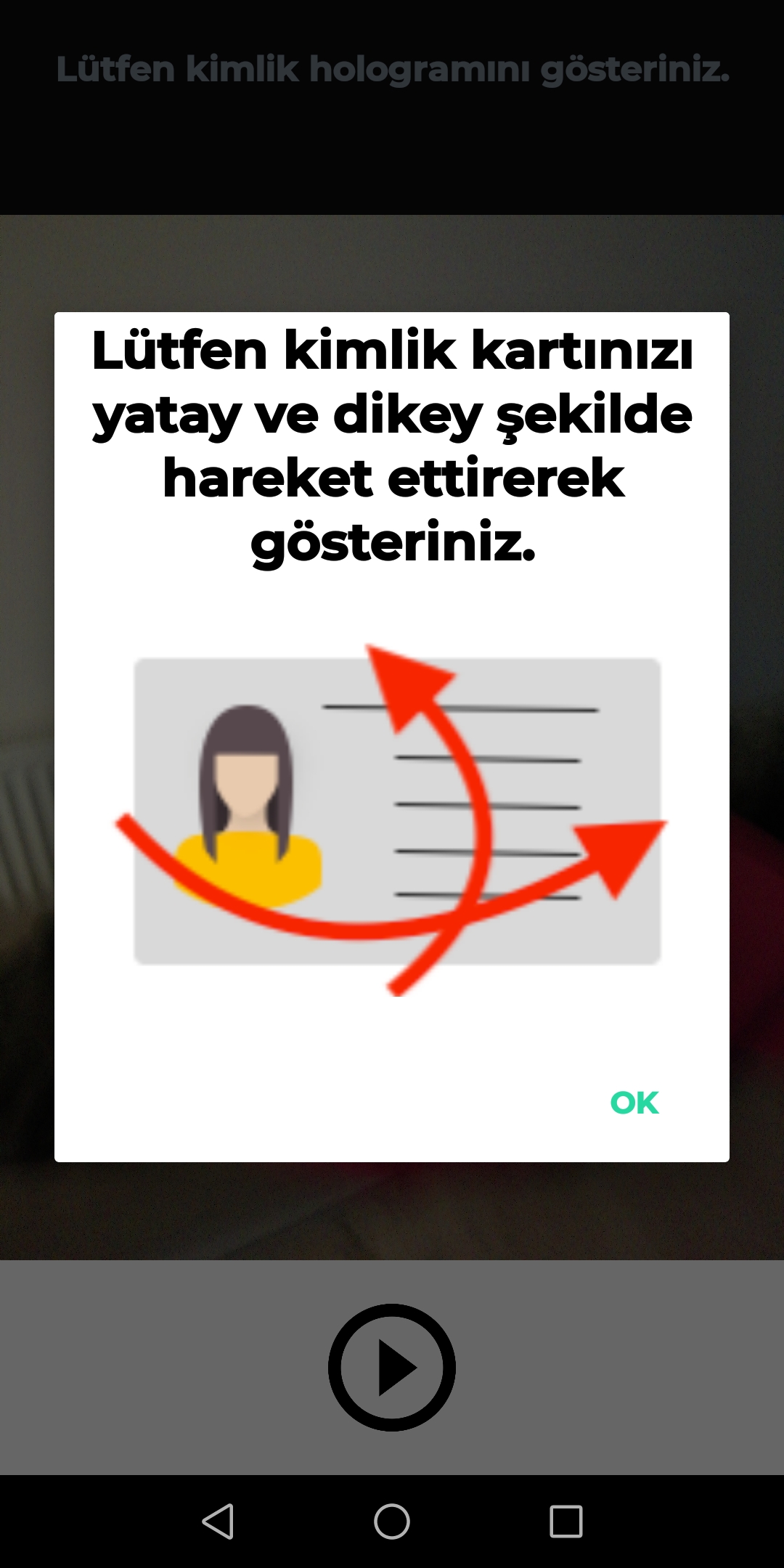
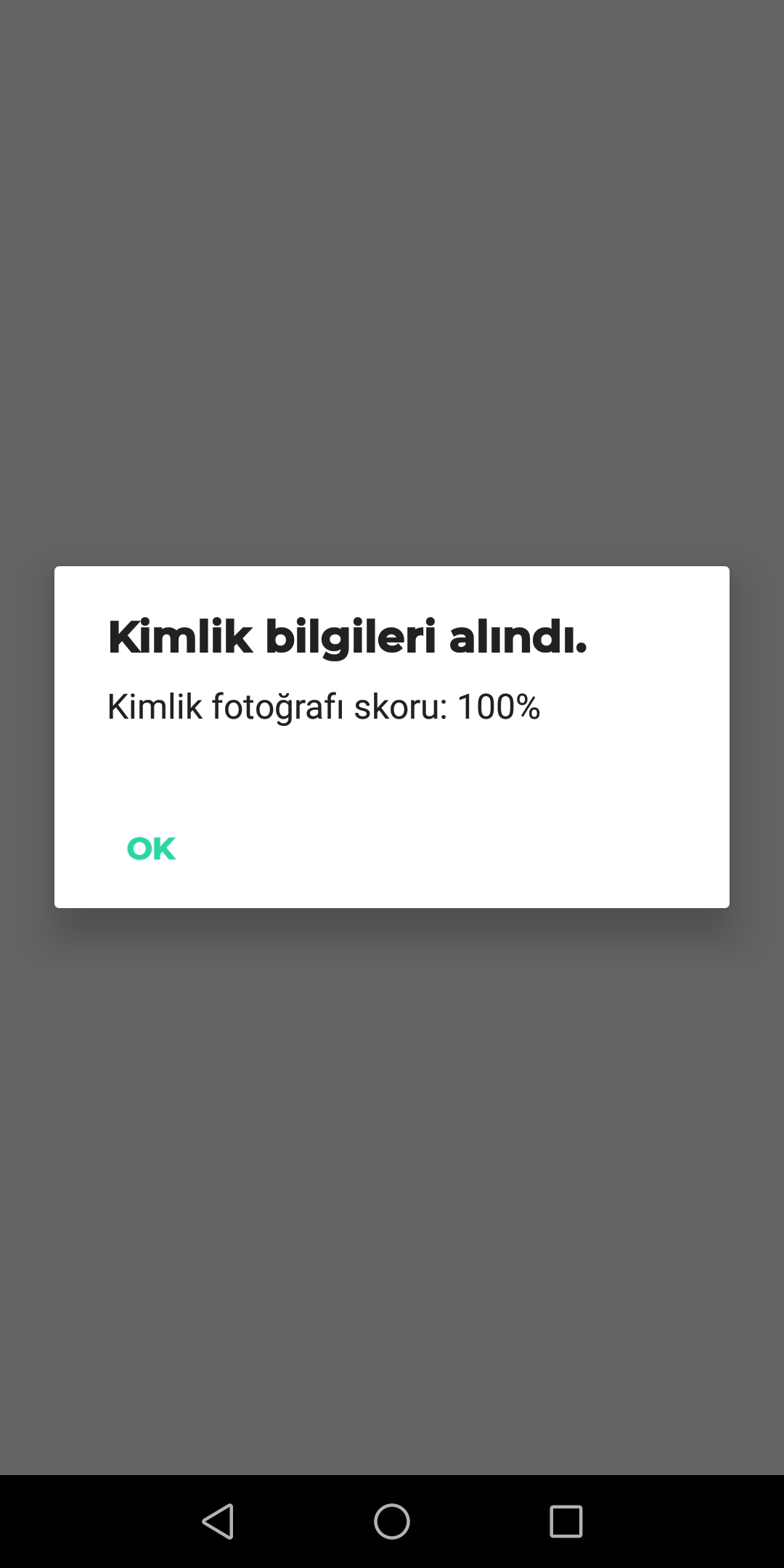
RainbowVerification("RAINBOWVERIFICATION"),

OCRNFCSimilarity("OCRNFCSIMILARITY"),

OCRMRZSimilarity("OCRMRZSIMILARITY");

}

Note: For a successful hologram control both HologramFaceVerification and HologramVerification should be passed in “controlResults”.



**API Call:**

**Server Suffix:** /id/hologram

**Type: POST**

**Input:**

{

video: String //captured hologram video, in base64 format

rotate: Int //clock-wise rotation to apply to the video before control, 0/90/180/270

cardType:String //“NEW-ID” for new id card, “NEW-DRIVER” for new drivers card

transactionId:String //transaction ID for the on-going transaction

}

**Output:**

HologramDetectionReturnModel, details listed above.



1. **Liveness Process**

****

Before starting the Liveness Process we should start the *Loader* and wait until it loads necessary components.

*import com.techsign.rkyc.util.Loader;*

*Loader.load(getBaseContext(), new Loader.LoaderListener() {*

*@Override*

*public void loaded() {*

*Log.i(TAG, "loaded");*

*}*

*});*

First, you need to add a fragment to the activity layout which will make the liveness control.

*<fragment*

*android:layout\_width="wrap\_content"*

*android:layout\_height="wrap\_content"*

*android:id="@+id/fragment\_liveness"*

*/>*

Then you need to decide which camera api will be used for liveness check.

*Boolean isCamera2supported = CameraUtil.isCamera2Supported(this,"1") // front camera*

If camera2 is not supported you should use *LivenessFragmentForCamera1*(), otherwise use *LivenessFragment*().

Then we set a *FragmentListener* and set a *FragmentListener* to it.

*public interface FragmentListener{*

*void onVideoCaptured(String videoPath); // video captured*

*void onVideoStarting(); //preparation time starting see VIDEO\_PREPARATION\_TIME\_IN\_SECONDS*

*void onVideoStarted(); // video recording started*

*void onVideoCanceled(); // video canceled*

*void onFaceDetectionFailure(boolean isAway); / face detection succeed but distance is far away if isAway is true, too close if isAway is false*

*void onFaceDetectionSuccess(); // face detection succeed and distance is proper*

*void onFaceDetectionIdle(); // no face detected*

*void onMultiFaceDetected(); func onMultiFaceDetected() // multiface detected*

*}*

Note: When the start() method called face detection process starts and when the user’s face is detected properly, preparation time starts and *onVideoStarting()* function is called. You can show a countdown timer on screen for a better user experience like in the demo application. If no face detected, multiface detected or distance is not proper relevant delegate function is called and preparation time resets. If the user’s face is properly detected upon the preparation time, video recording starts and *onVideoStarted()* function is called.

Note 2: Face detection layer (default layer is dashed rectangle) is configurable. See “*Customization”* sectionat the bottom of the document.

Then you can start the process by calling *start()*. When the video is captured we will receive a callback to *onVideoCaptured* with a videoPath. You can get the base64 of the video as below from this path and the rotate parameter from *livenessFragment*

*File file = new File(videoPath);*

*int size = (int) file.length();*

*byte[] bytes = new byte[size];*

*try {*

*BufferedInputStream buf = new BufferedInputStream(new FileInputStream(file));*

*buf.read(bytes, 0, bytes.length);*

*buf.close();*

*} catch (FileNotFoundException e) {*

*Log.e(TAG, "sendVideo: video file not found ");*

*} catch (IOException e) {*

*Log.e(TAG, "sendVideo: video file IO Exception: " + e.getMessage());*

*}*

*String base64 = Base64.encodeToString(bytes,Base64.DEFAULT);*

*int rotate = fragmentLiveness.rotate;*

Liveness service can be used by calling ServerCall.checkLiveness() function which requires a FaceVerificationModel and a TechSignServiceListener<VerifyFaceReturnModel>.

FaceVerificationModel consists of three fields,

* String “video” => base64 of recorded selfie video[1]
* String “transactionId” => gathered from card process
* int “rotate” => rotation of recorded video (0,90,180 or 270)

[1]. The recorded video should be around 5 seconds and 480p.

If the service call is successful, onSuccess method called with a VerifyFaceReturnModel which consist of,

* String “id” => citizen id
* String “name” => user’s name
* String “surname” => user’s surname
* String “birthDate” => user’s birth date
* String “expireDate” => user’s expire date
* String “picture” => base64 of user’s picture on identity card which he/she used for registration
* ControlEntryModel[] “controlResults” => ControlEntryModel consists of,
* String “entryType” => ControlEntryTypes enum[2]
* Float “entryScore” => the result of the control between 0 and 1.
* Boolean “entryInterpretation” => true if the control is successful
* String “controlId” => unique identifier of the control

[2].public enum ControlEntryTypes {

PhotoCheatVerification("PHOTOCHEAT"),

HologramVerification("HOLOGRAM"),

HologramFaceVerification("HOLOGRAMFACE"),

FaceLivenessVerification("FACELIVENESS"),

FaceVerification("FACE"),

HiddenPhotoVerification("HIDDENPHOTO"),

SignaturePhotoVerification("SIGNATUREPHOTO"),

GuillocheVerification("GUILLOCHEVERIFICATION"),

RainbowVerification("RAINBOWVERIFICATION"),

OCRNFCSimilarity("OCRNFCSIMILARITY"),

OCRMRZSimilarity("OCRMRZSIMILARITY");

}

Note: For a successful login both *“*FaceVerification*”* and *“*FaceLivenessVerification*”* in “controlResults” should be passed.

**API Call:**

**Server Suffix:** /face/verify

**Type: POST**

**Input:**

{

transactionId:String //transaction ID for the on-going transaction

video: String //captured liveness video, in Base64 format

rotate: Int //clock-wise rotation to apply to the video before control, 0/90/180/270

}

**Output:**

VerifyFaceReturnModel, details listed above.



We use camera2 api on the sample app(rkyc-android-app) for video recording,

see <https://github.com/android/camera-samples/tree/master/Camera2VideoJava>

**Customization**

**OCR**

You can change the guide color and scan instruction of *IDCardReader* as below.

*idCardReader.setCustomGuideColor(Color.RED);*

*idCardReader.setCustomScanInstruction("TEST SCAN INSTRUCTION");*

You can set a custom capture view which will be rendered on the top of default card detection view according to orientation and you can set a cancel button id if you have a cancel button on the view.

*idCardReader.setCustomCardCaptureView(<CUSTOM\_CAPTURE\_VIEW>)*

*IDCardReader.setCustomCancelButtonId(<CANCEL\_BUTTON\_ID>)*

**Liveness**

You can change default face detection layers shape like in the below.

*LivenessFragment.FACE\_DETECTION\_IDLE\_LAYER = <CUSTOM FACE DETECTION IDLE LAYER>*

*LivenessFragment.FACE\_DETECTION\_SUCCESS\_LAYER = <CUSTOM FACE DETECTION SUCCESS LAYER>*

*LivenessFragment.FACE\_DETECTION\_FAIL\_LAYER = <CUSTOM FACE DETECTION FAIL LAYER>*

Default layers are in the below.

*if(FACE\_DETECTION\_FAIL\_LAYER == null) {*

*FACE\_DETECTION\_FAIL\_LAYER = new GradientDrawable();*

*FACE\_DETECTION\_FAIL\_LAYER.setCornerRadius(10);*

*FACE\_DETECTION\_FAIL\_LAYER.setStroke(5, Color.RED, 100, 100);*

*}*

*if(FACE\_DETECTION\_SUCCESS\_LAYER == null) {*

*FACE\_DETECTION\_SUCCESS\_LAYER = new GradientDrawable();*

*FACE\_DETECTION\_SUCCESS\_LAYER.setCornerRadius(10);*

*FACE\_DETECTION\_SUCCESS\_LAYER.setStroke(5, Color.GREEN, 100, 100);*

*}*

*if(FACE\_DETECTION\_IDLE\_LAYER == null) {*

*FACE\_DETECTION\_IDLE\_LAYER = new GradientDrawable();*

*FACE\_DETECTION\_IDLE\_LAYER.setCornerRadius(10);*

*FACE\_DETECTION\_IDLE\_LAYER.setStroke(5, Color.WHITE, 100, 100);*

*}*