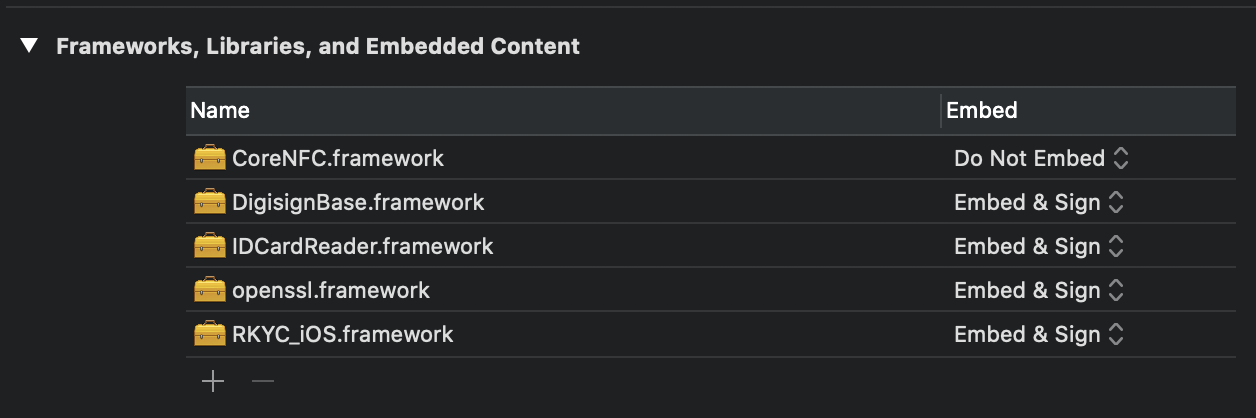
**iOS**

* Supported minimum sdk version is iOS 8.0 which can run on approximately 99.9% of devices according to Apple App Store.

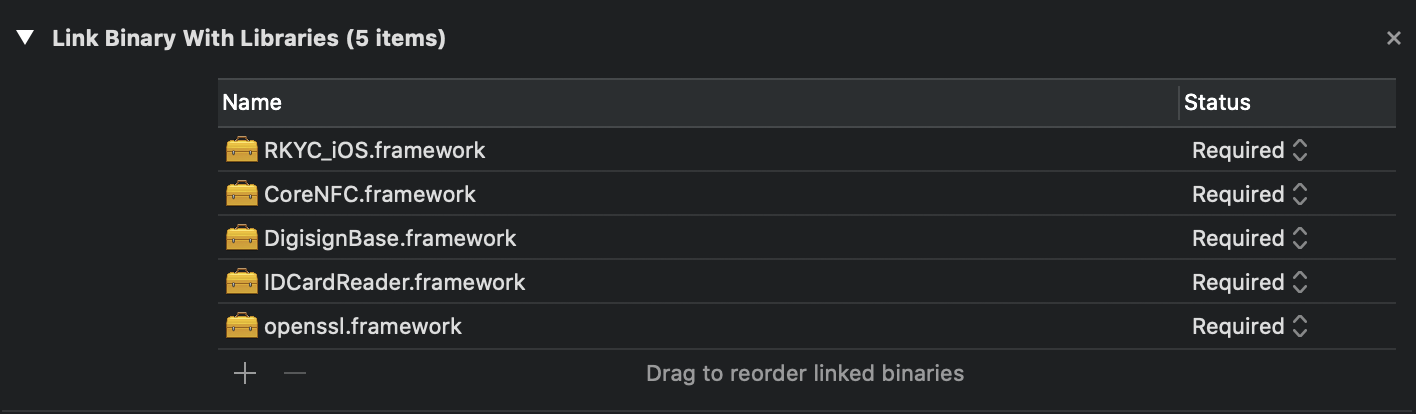
**Integration**

There are 4 frameworks namely DigisinBase, IDCardReader, openssl and RKYC\_iOS. These frameworks should be added to the project folder and the folder path should be added to Build Settings -> Framework Search Paths. After that Frameworks, Libraries and Embedded Content in the General tab should look like this;

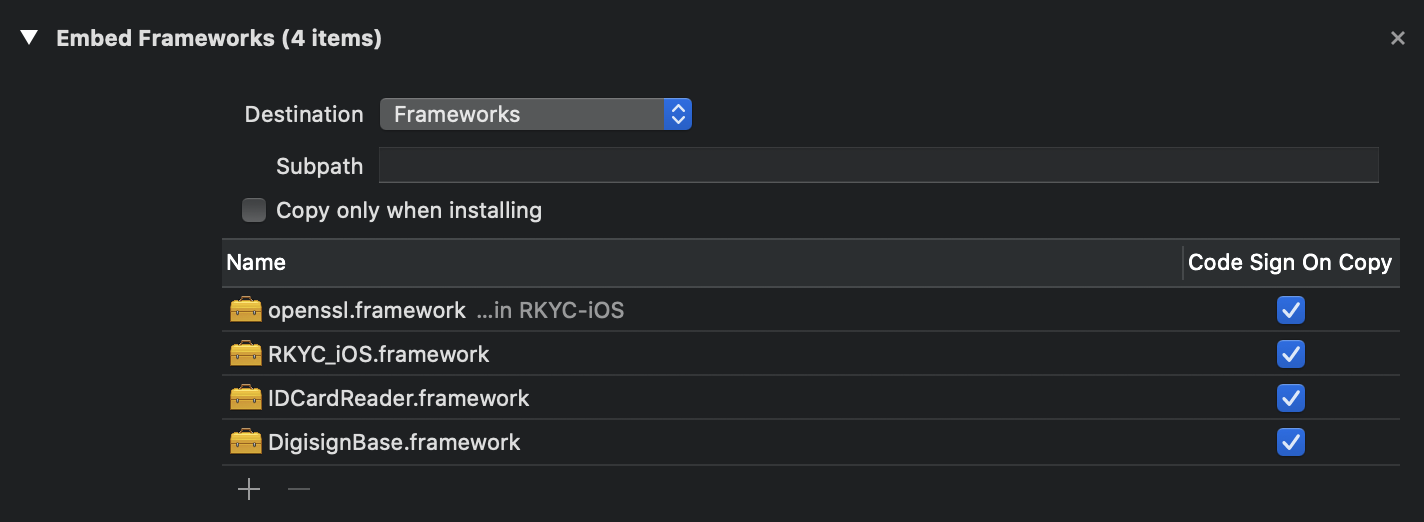


We also add the iOS CoreNFC framework for the use of NFC.

After that, Build Phases -> Link Binary With Libraries should look like this;

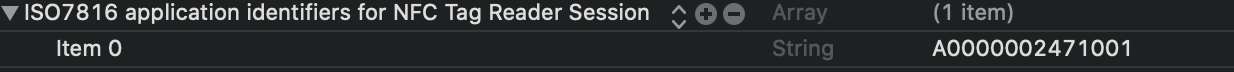


And Build Phases -> Embed Frameworks should look like this;

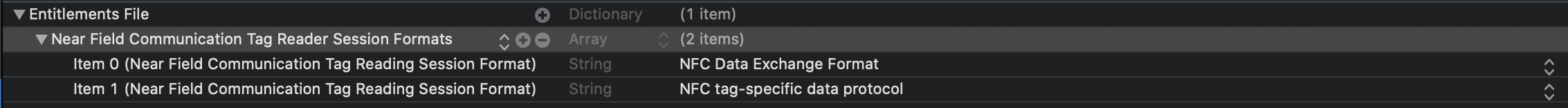


**NFC Configurations**

1-) The below key value pairs should be added to the info.plist file.



2-) A Code Signing Entitlements file should be added to project => Build Settings

3-) The Code Signing Entitlements file should contain below key value pairs.

**API**

Test API Root URL:

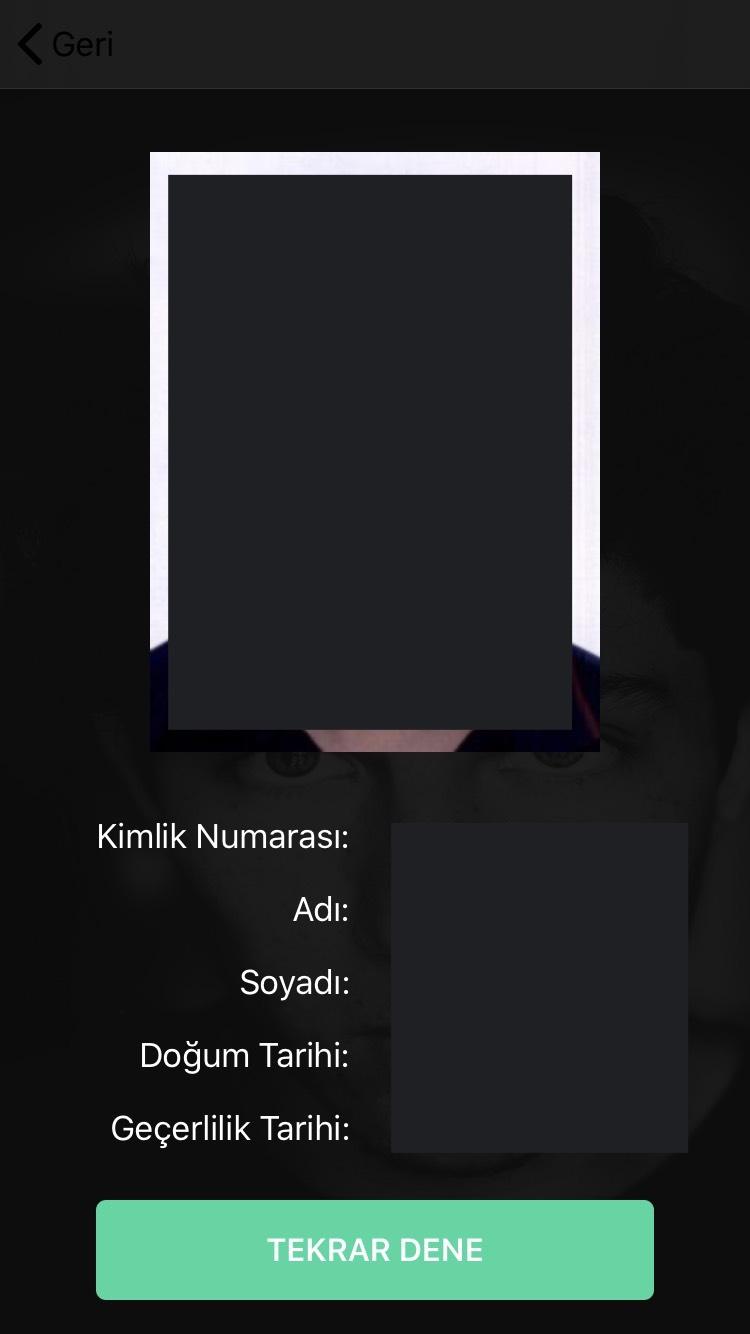
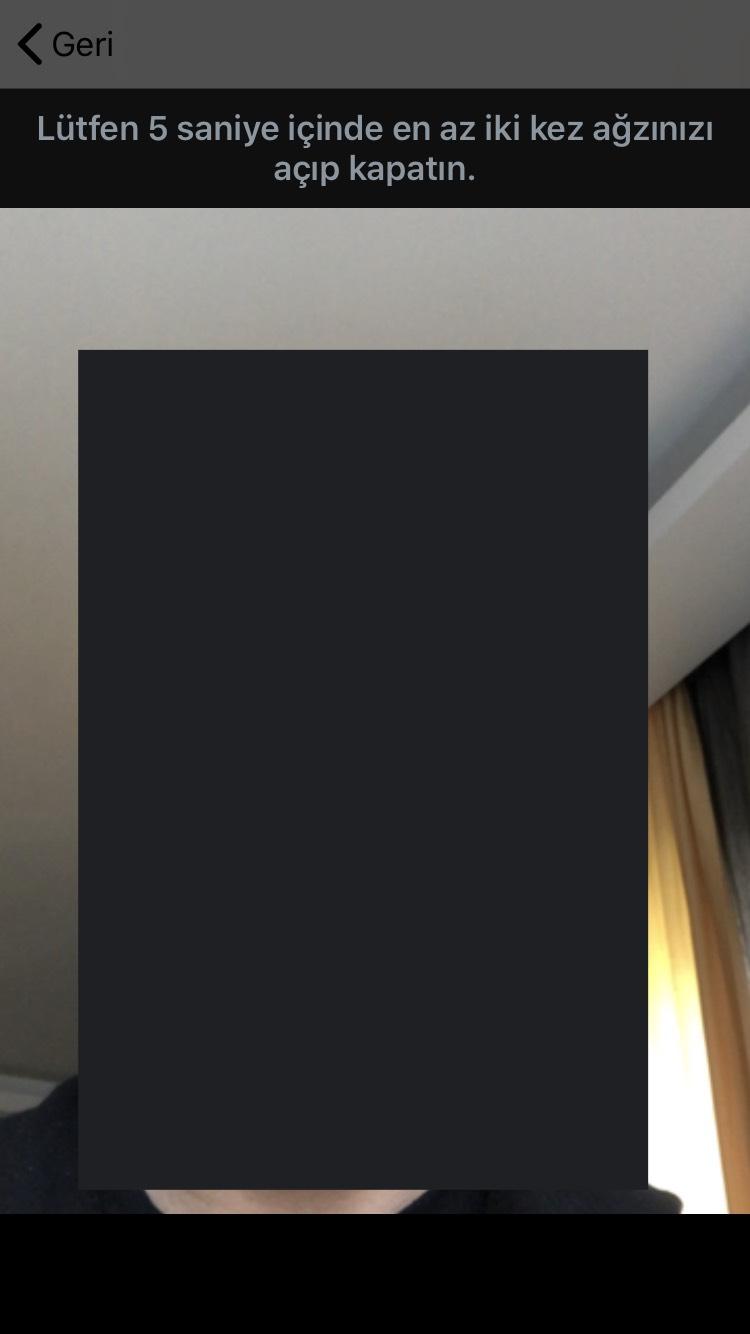
<http://169.62.55.86:8098/>

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 (Result -> Void) function 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 LoginModel and a Result<VerifyFaceReturnModel,NSError>.

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 : String{

case PhotoCheatVerification = "PHOTOCHEAT"

case HologramVerification = "HOLOGRAM"

case HologramFaceVerification = "HOLOGRAMFACE"

case FaceLivenessVerification = "FACELIVENESS"

case FaceVerification = "FACE"

case HiddenPhotoVerification = "HIDDENPHOTO"

case SignaturePhotoVerification = "SIGNATUREPHOTO"

case GuillocheVerification = "GUILLOCHEVERIFICATION"

case RainbowVerification = "RAINBOWVERIFICATION"

case OCRNFCSimilarity = "OCRNFCSIMILARITY"

case 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

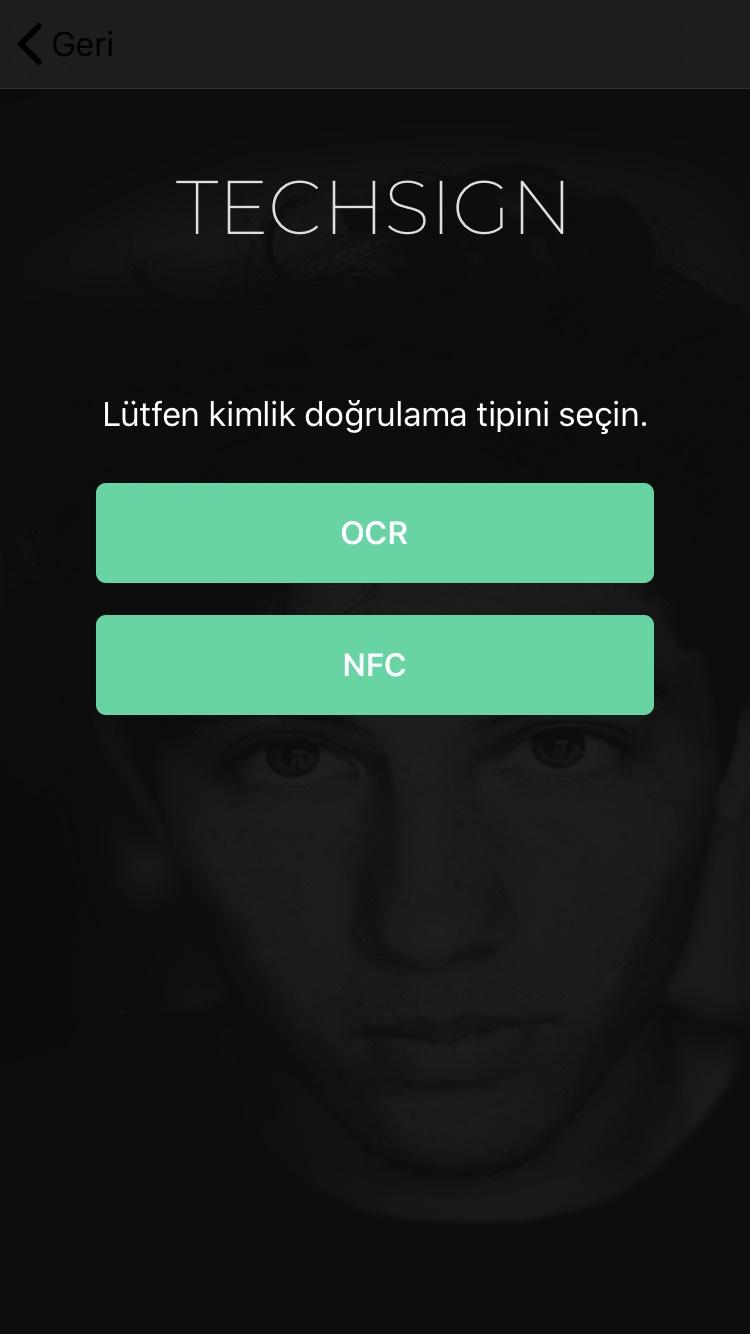
rotate: Int //clock-wise rotation to apply to the video before control, 0/90/180/270

}

**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.

To get the result from *“IDCardReader”*  you need add *“IDCardReaderDelegate”* to your class definition and implement its methods shown in below.

*func onCanceled(controller:IDCardReader)*

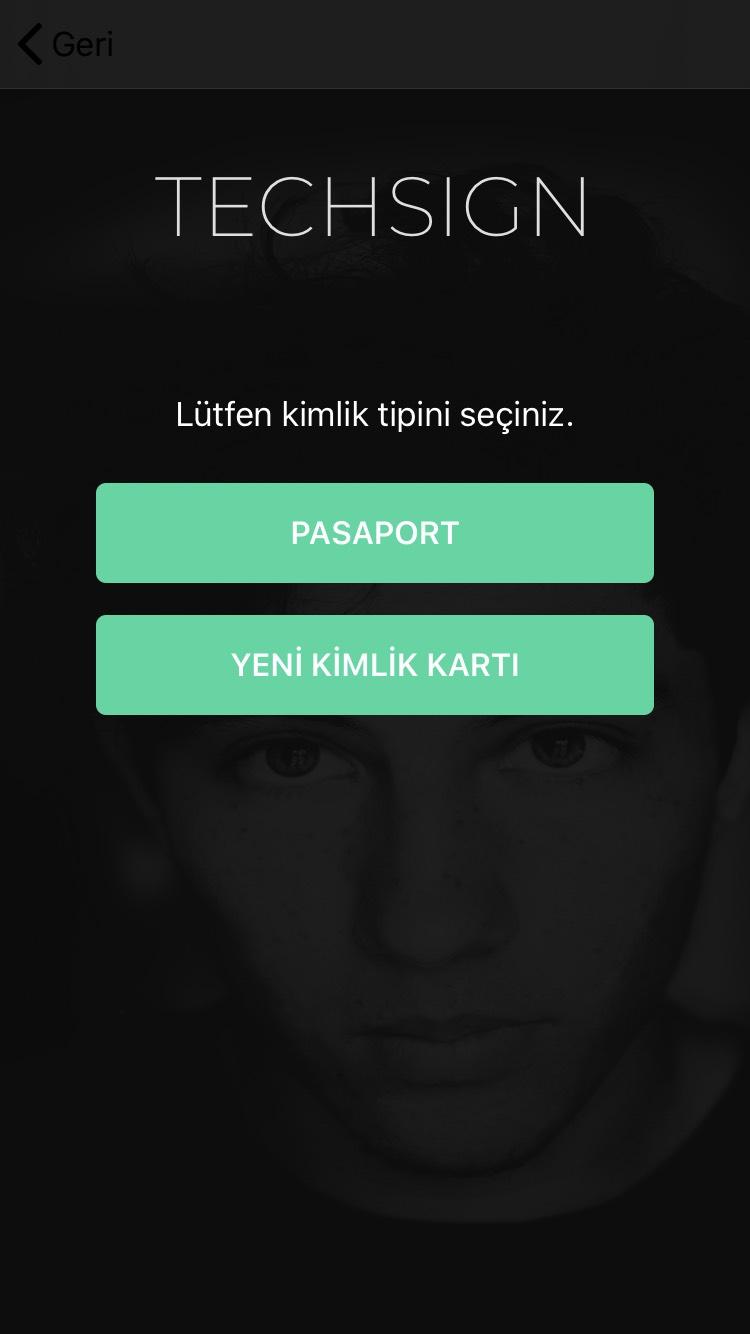
*func onProvided(cardModel:IDCardModel,controller:IDCardReader)*

Then, initialize the IDCardReader like below.

*let idCardReader = IDCardReader(viewController:self, delegate:self)*

There are two methods for registering, OCR and NFC.

* **With NFC**

****

Set below parameters to the *“IDCardReader”.*

*idCardReader.setCardType(CARD\_TYPE*[1]*)*

*idCardReader.setIsMrzActive(true)*

*idCardReader.setSkipOcr(true)*

Present *idCardReader* using “*present”* function.

*idCardReader.present(modalPresentationStyle:UIModalPresentationStyle, animated:Bool, completion: (() -> Void)?)*

[1]. The parameter *“CARD\_TYPE”* is an enum from *“IDCardType”.* Which consists of

* UNKNOWN
* PASSPORT
* NEW\_TC\_ID
* OLD\_TC\_ID
* OLD\_TC\_DRIVER\_LICENSE
* DRIVER\_LICENSE
* NEW\_TC\_ID\_BACK
* DRIVER\_BACK

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

When the card photo is taken the card detection process starts. When the card detection process is finished, we receive a callback to *“onProvided”.* Which we implemented above.

We can get the card image from this callback and convert it to base64 format;

*let base64 = cardModel.cardImage!.jpegData(compressionQuality: 1.0)!.base64EncodedString()*

Then we make an API call for OCR. (ServerCall.verifyMrzOcr)

**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)

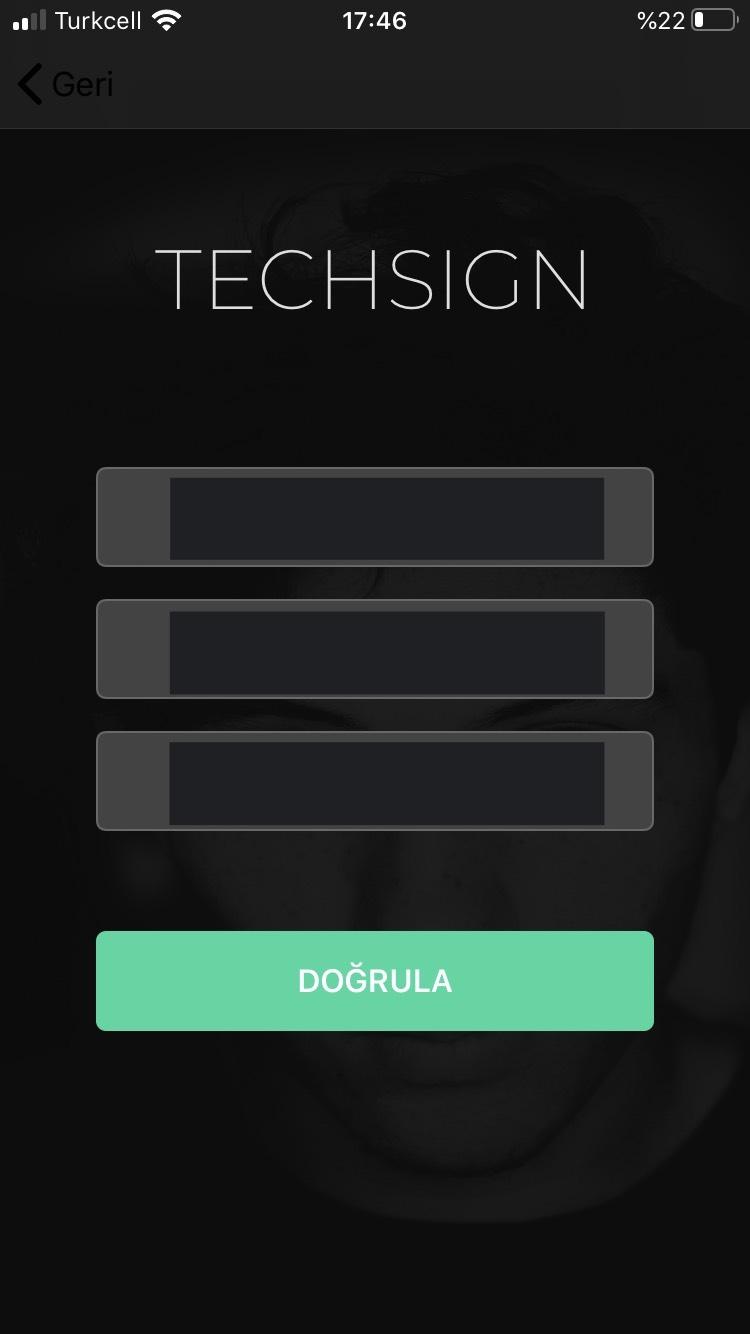
name:String => Name

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.

*if NFCNDEFReaderSession.readingAvailable, #available(iOS 13.0, \*){*

*//device has NFC*

*}else{*

*//device has not NFC*

*}*

NFC verification requires document number, birth date and expire date. All of these are in *“*MrzOcrOutputModel*”* from *“*ServerCall.verifyMrzOcr)*”.* Note that the dates of *“*MrzOcrOutputModel*”* 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 create a *NFCReader*;

*let reader = NFCReader()*

* We need to create a mrz key from birthdate, expire date and document no;

*var mrzKey = ""*

*do{*

*mrzKey = try reader.getMrzKey(<document no>, <birth date>,<expire date>)*

*}catch{*

*// entered information is wrong*

*}*

* To start verification and get “*ValidationInputModel”* which will be used to register user,

*reader.read(mrzKey: mrzKey, type: <NFCType.IDCARD or NFCType.PASSPORT>, completed: { (model,error) in*

*if(error != nil){*

*// error*

*}else{*

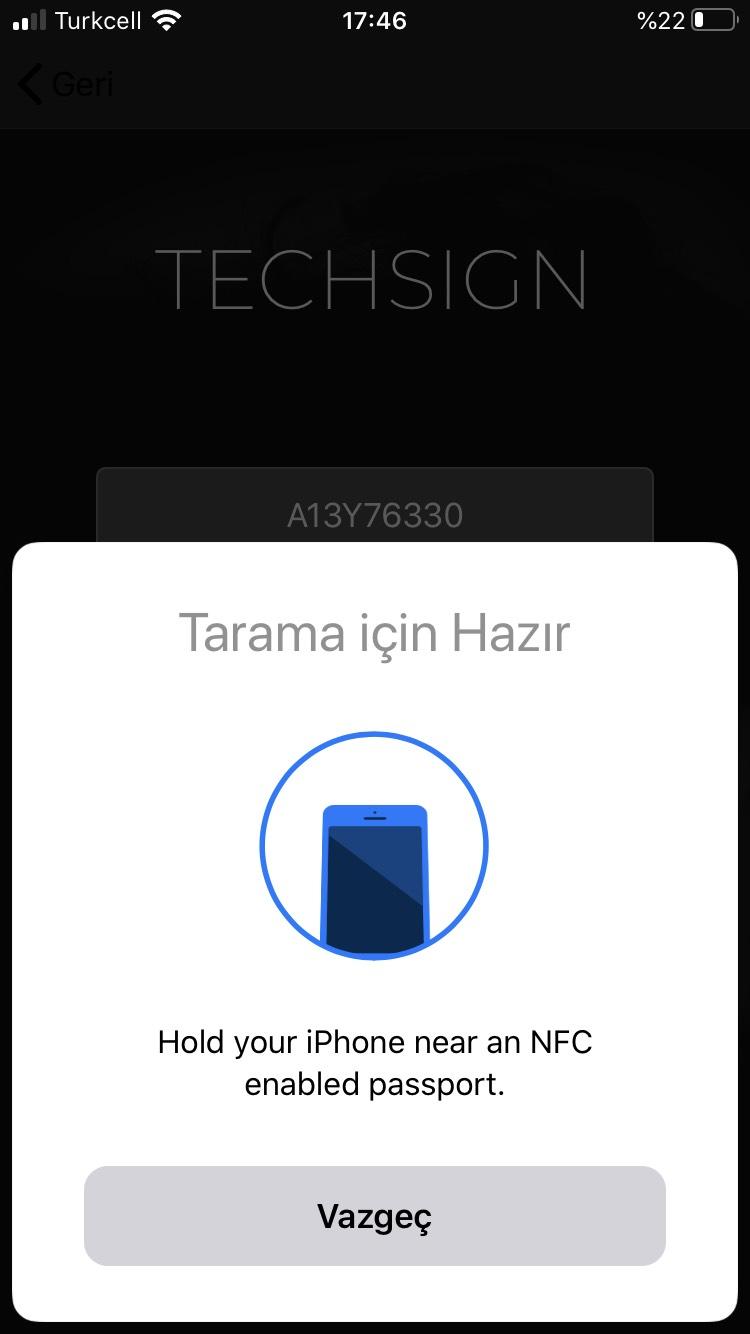
*let validationInfo = IDValidationModel(dg1: model!.dg1, dg2: model!.dg2, sod: model!.sod, dg11: model!.dg11, dg12: model!.dg12, dg14: model!.dg14, dg15: model!.dg15)*

*let validationModel = ValidationInputModel(validationInformation: validationInfo,*

*idType: self.idType!)*

*}*

*}*

**

Then using the model “*ValidationInputModel*” for ServerCall.validateAndSaveID(), 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[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 : String{

case PhotoCheatVerification = "PHOTOCHEAT"

case HologramVerification = "HOLOGRAM"

case HologramFaceVerification = "HOLOGRAMFACE"

case FaceLivenessVerification = "FACELIVENESS"

case FaceVerification = "FACE"

case HiddenPhotoVerification = "HIDDENPHOTO"

case SignaturePhotoVerification = "SIGNATUREPHOTO"

case GuillocheVerification = "GUILLOCHEVERIFICATION"

case RainbowVerification = "RAINBOWVERIFICATION"

case OCRNFCSimilarity = "OCRNFCSIMILARITY"

case OCRMRZSimilarity = "OCRMRZSIMILARITY"

}

**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

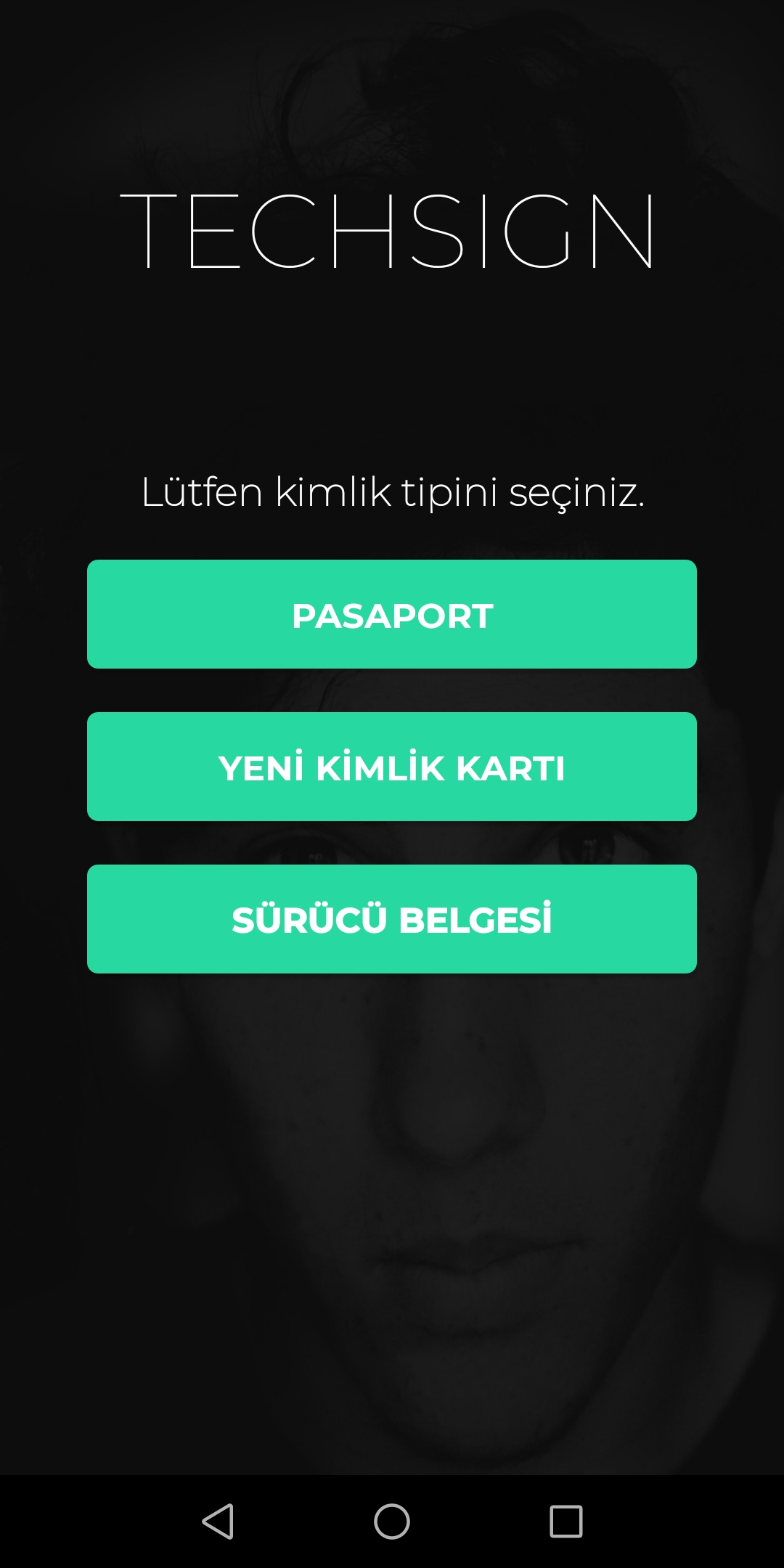
transactionId:String // if the transaction started with ocr process before nfc process the transactionId of the ocr process otherwise null

}

**Output:**

*SaveIDReturnModel*, details listed above.

* **With OCR**

****

Set below parameters to the *“IDCardReader”.*

*idCardReader.setCardType(CARD\_TYPE*[1]*)*

*idCardReader.setIsMrzActive(false)*

*idCardReader.setSkipOcr(true)*

Present *idCardReader* using “*present”* function.

*idCardReader.present(modalPresentationStyle:UIModalPresentationStyle, animated:Bool, completion: (() -> Void)?)*

[1]. The parameter *“CARD\_TYPE”* is an enum from *“IDCardType”.* Which consists of

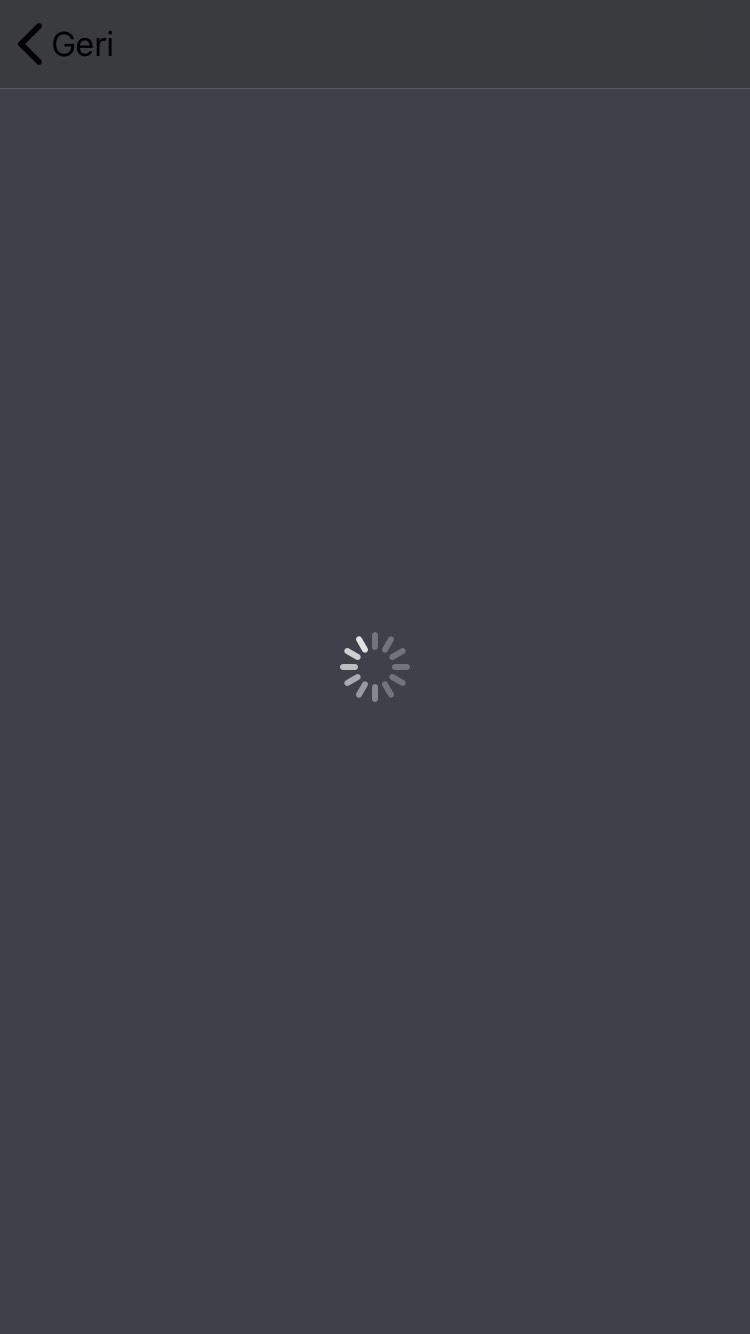
* UNKNOWN
* PASSPORT
* NEW\_TC\_ID
* OLD\_TC\_ID
* OLD\_TC\_DRIVER\_LICENSE
* DRIVER\_LICENSE
* NEW\_TC\_ID\_BACK
* DRIVER\_BACK

For the OCR case, we should use PASSPORT , NEW\_TC\_ID or DRIVER\_LICENSE.

When the card image is captured the OCR process starts. When the card image is captured, we receive a callback to *“onProvided”.* Which we implemented above. We can take the image from *“IDCardModel”*

*let image:UIImage = idCardModel.cardImage!*

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

**

All of the calls require a CheckIDModel and a Result<CheckIdReturnModel,NSError>.

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

*let base64 = cardModel.cardImage!.pngData()!.base64EncodedString()*

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 : String{

case PhotoCheatVerification = "PHOTOCHEAT"

case HologramVerification = "HOLOGRAM"

case HologramFaceVerification = "HOLOGRAMFACE"

case FaceLivenessVerification = "FACELIVENESS"

case FaceVerification = "FACE"

case HiddenPhotoVerification = "HIDDENPHOTO"

case SignaturePhotoVerification = "SIGNATUREPHOTO"

case GuillocheVerification = "GUILLOCHEVERIFICATION"

case RainbowVerification = "RAINBOWVERIFICATION"

case OCRNFCSimilarity = "OCRNFCSIMILARITY"

case 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.

**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 : String{

case PhotoCheatVerification = "PHOTOCHEAT"

case HologramVerification = "HOLOGRAM"

case HologramFaceVerification = "HOLOGRAMFACE"

case FaceLivenessVerification = "FACELIVENESS"

case FaceVerification = "FACE"

case HiddenPhotoVerification = "HIDDENPHOTO"

case SignaturePhotoVerification = "SIGNATUREPHOTO"

case GuillocheVerification = "GUILLOCHEVERIFICATION"

case RainbowVerification = "RAINBOWVERIFICATION"

case OCRNFCSimilarity = "OCRNFCSIMILARITY"

case 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

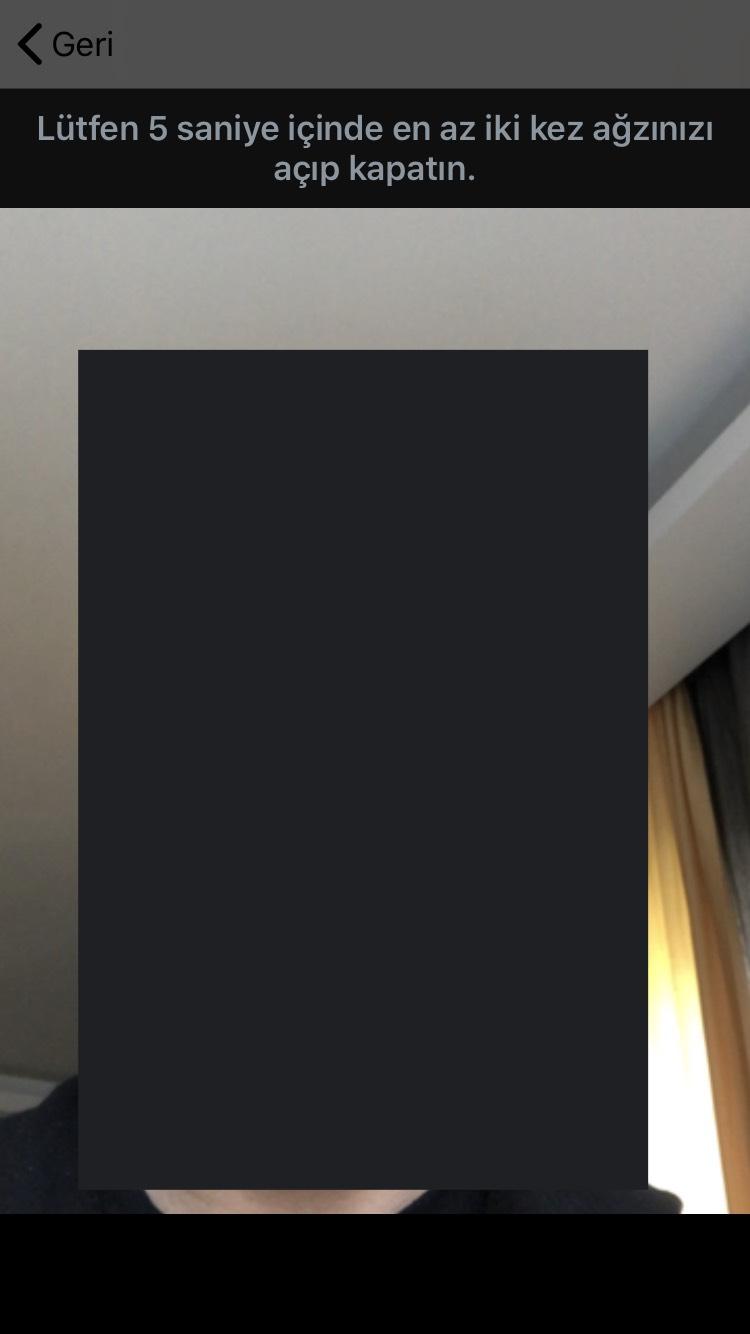
transactionId:String //transaction ID for the on-going transaction

}

**Output:**

HologramDetectionReturnModel, details listed above.

1. **Liveness Process**



First *‘LivenessDelegate’* should be implemented in a ViewController that will use *‘LivenessController’* .

*public protocol LivenessDelegate {*

*func onVideoCaptured(videoURL:URL,rotate:Int) // video captured*

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

*func onVideoStarted() // video recording started*

*func onVideoCanceled() // video canceled*

*func onFaceDetectionFailure(isAway: Bool) // face detection succeed but distance is far away if isAway is true, too close if isAway is false*

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

*func onFaceDetectionIdle() // no face detected*

*func onMultiFaceDetected() // multiface detected*

*}*

Then *‘LivenessController’* should be added as a child ViewController.

*var livenessController = LivenessController()*

*livenessController.setDelegate(self) // we set the delegate implemented before.*

*livenessContainer.addSubview(livenessController.view) // add the livenessController’s view to another view (like a container, which has a adjusted position and frame) as subview*

*livenessController.start() // start the controller, when the onVideoCaptured() callback called controller stops, you need to run this call for restart*

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.

When *func onVideoCaptured(videoURL:URL,rotate:Int)* called video capture is finished. You can take the base64 of the video as below.

*let fileData = try? Data(contentsOf: outputURL)*

*let base64 = fileData!.base64EncodedString()*

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

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 : String{

case PhotoCheatVerification = "PHOTOCHEAT"

case HologramVerification = "HOLOGRAM"

case HologramFaceVerification = "HOLOGRAMFACE"

case FaceLivenessVerification = "FACELIVENESS"

case FaceVerification = "FACE"

case HiddenPhotoVerification = "HIDDENPHOTO"

case SignaturePhotoVerification = "SIGNATUREPHOTO"

case GuillocheVerification = "GUILLOCHEVERIFICATION"

case RainbowVerification = "RAINBOWVERIFICATION"

case OCRNFCSimilarity = "OCRNFCSIMILARITY"

case OCRMRZSimilarity = "OCRMRZSIMILARITY"

}

Note: For a successful register both *“faceResult”* and *“livenessResult”* should be true.

**API Call:**

**Server Suffix:** /face/verify

**Type: POST**

**Input:**

{

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

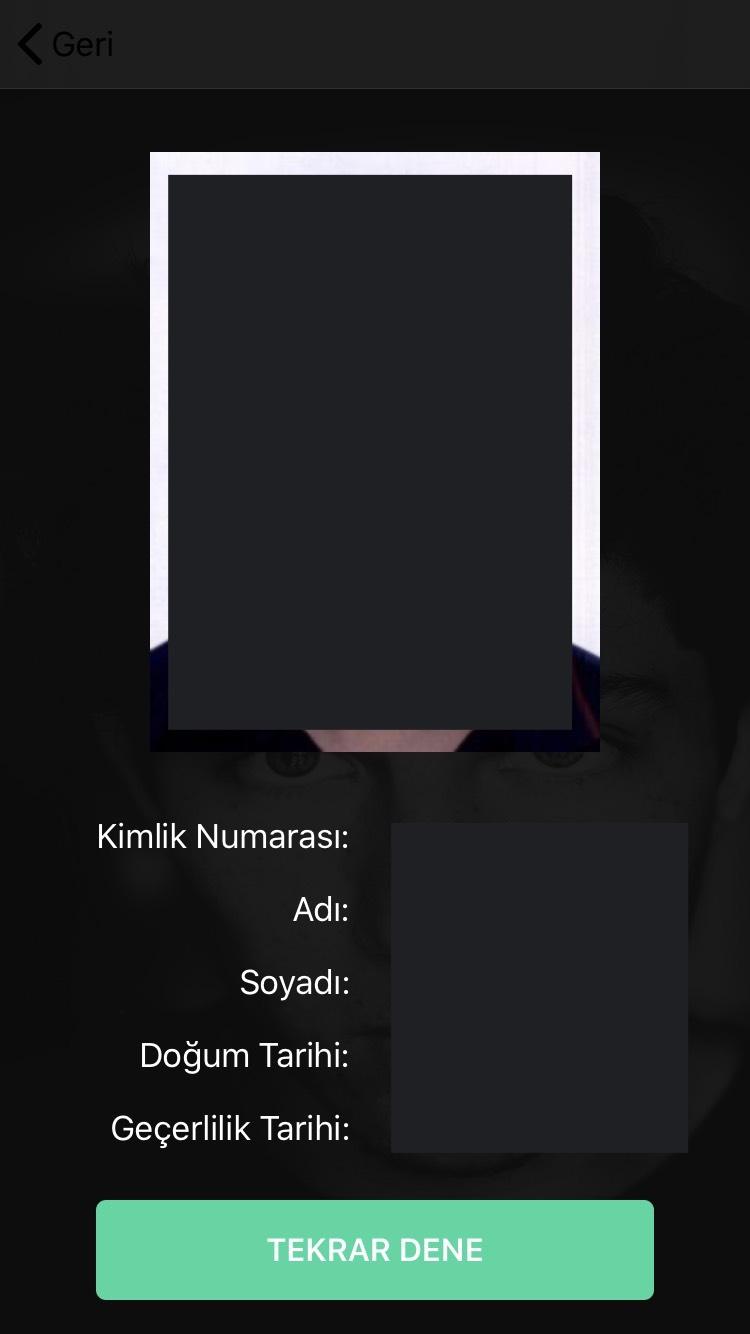
video: String //captured liveness video, in Base64 format

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

}

**Output:**

VerifyFaceReturnModel, details listed above.



We use AVFoundation api on the sample app(RKYC-iOS-Test-App) for video recording,

see <https://stackoverflow.com/a/41698917/4824165>

**Customization**

**OCR**

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

*idCardReader.setCustomGuideColor(UIColor.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 tag if you have a cancel button on the view.

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

*idCardReader.setCustomCancelButtonTag(<CANCEL\_BUTTON\_TAG>)*

You can disable the default cancel button, if you have one in the custom view.

*idCardReader.setIsCancelButton(false)*

**NFC**

You can customize the NFC reading screen by using the *customDisplayMessage* parameter of *NFCReader’s* *read* function. You can find a *customDisplayMessage* sample below.

*private var customDisplayMessage:((NFCViewDisplayMessage) -> String?)?*

*customDisplayMessage = { (NFCViewDisplayMessage) -> String in switch NFCViewDisplayMessage {*

*case .requestPresentPassport:*

*return "Hold your iPhone near an NFC enabled passport."*

*case .authenticatingWithPassport(let progress):*

*let progressString = self.handleProgress(percentualProgress: progress)*

*return "Authenticating with passport.....\n\n\(progressString)"*

*case .readingDataGroupProgress(let dataGroup, let progress):*

*let progressString = self.handleProgress(percentualProgress: progress)*

*return "Reading \(dataGroup).....\n\n\(progressString)"*

*case .error(let tagError):*

*switch tagError {*

*case TagError.TagNotValid:*

*return "Tag not valid."*

*case TagError.MoreThanOneTagFound:*

*return "More than 1 tags was found. Please present only 1 tag."*

*case TagError.ConnectionError:*

*return "Connection error. Please try again."*

*case TagError.InvalidMRZKey:*

*return "MRZ Key not valid for this document."*

*case TagError.ResponseError(let description):*

*return "Sorry, there was a problem reading the passport. \(description)"*

*default:*

*return "Sorry, there was a problem reading the passport. Please try again"*

*}*

*case .successfulRead:*

*return "Passport read successfully"*

*}*

*}*

*func handleProgress(percentualProgress: Int) -> String {*

*let p = (percentualProgress/20)*

*let full = String(repeating: "🟢 ", count: p)*

*let empty = String(repeating: "⚪️ ", count: 5-p)*

*return "\(full)\(empty)"*

*}*

Note: We can’t change the label above the screen (“Ready to Scan” for English, “Tarama için Hazır” for Turkish), because it is not allowed by Apple’s SDK.

**Liveness**

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

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

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

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

Default layers are in the below.

*private static func createDefaultIdleLayer() -> CAShapeLayer {*

*let layer = CAShapeLayer()*

*layer.strokeColor = UIColor.white.cgColor*

*layer.lineDashPattern = [100, 100]*

*layer.fillColor = nil*

*return layer*

*}*

*private static func createDefaultFailLayer() -> CAShapeLayer {*

*let layer = CAShapeLayer()*

*layer.strokeColor = UIColor.red.cgColor*

*layer.lineDashPattern = [100, 100]*

*layer.fillColor = nil*

*return layer*

*}*

*private static func createDefaultSuccessLayer() -> CAShapeLayer {*

*let layer = CAShapeLayer()*

*layer.strokeColor = UIColor.green.cgColor*

*layer.lineDashPattern = [100, 100]*

*layer.fillColor = nil*

*return layer*

*}*