Accept SDK

Android version

Release December 2015

Author: Wirecard Technologies GmbH

www.wirecard.com | info@wirecard.com

For internal use only

Contents

1 Introduction

This document describes the Accept SDK pertaining to Android mobile devices, with a specific focus on PosMate Chip'n'Pin devices. This document is release 1.3.4 on 12th of November 2014.

1.1 Audience

This document is intended for Consumer Cards Platform developers and stakeholders.

1.2 Document Conventions

This document has been created by the Wirecard Consumer Cards Team. It is confidential and intended for internal use only.

1.3 Related Documents

1.4 Revision History

SDK Version	Date	Name	Comments
1.4.9	08.01.2016	Patrik M.	Updated section 3.4: added loading of AID configuration after success initialization/login
1.4.8	29.09.2015	Marek H.	Changes in section 4. Chip&pin and Swiper devices support Added: 4.3.3 Swiper 4.3.4 OnSwipeDetectedListener 4.4.7 DisplayText 4.4.8 EmvError 4.4.9 TransactionResult 4.6 Events handling and communication with Swiper 4.6.1 Example of OnSwipeDetectedListener implementation (Swiper) 4.9 Processing payment with Swiper (chip&sign) 4.9.1 Accessing Swiper instance 4.9.2 Tip and gratuity 4.9.3 Connection 4.9.4 Autoconfiguration 4.9.5 MRC vs. start EMV 4.9.6 Configuration update 4.9.7 Preauthorization 4.9.8 Signature placing 4.9.9 Transaction finalization (capture)
1.3.4	04.09.2015	Patrik M.	Added section 3.11 Receipt containing description of ReceiptBuilder methods
1.3.4	12.11.2014	Damian K.	Added support for multi-currency

1.3.3	13.10.2014	Damian K.	Added WisePad terminal integration. Added support for the latest uEMVSwiper vendor SDK (ver. 2.4.0) Application cryptogram is sent with signature update. Added method to trigger "Forgot User Id" verification flow
1.3.2	06.08.2014	Damian K.	Added guide for Accept SDK integration with Android Studio build environment
1.3.1	31.07.2014	Damian K.	Accept SDK supports magstripe+PIN transaction flows
1.3.0	03.03.2014	Richard Tatham	Add section on API v3 Payment flow
1.2.1		Adam K.	Added section 4.7 containing diagrams presenting connection and transaction flows.
1.2.1		Adam K.	Added section 3.10 providing information about session handling.
1.2.1		Adam K.	Added new CNP related getters to Payment model.
1.2.1		Adam K.	Added desccriptions of following helper methods: getSupportedSwipers, getSupportedCNPControllers, loadCNPControllerByName, loadSwiperByName, getPrefTaxationEnabled, setPrefTaxationEnabled, clearSignature
1.2.1		Adam K.	Added 4.5.3 subsection showing example implementation of CNPListener interface
1.2.1		Adam K.	Added 4.2.1 subsectioncontaining information about mock configuration profiles.
1.2		Adam K.	Added section 3.9, describing method for sending transaction logs. This feature should shorten time spent on resolving issues related to transactions processing
1.2		Adam K.	Added section 2.3 showing how to add ThyronExtension to project
1.2		Adam K.	Added descriptions of new methods: getGratuityType, getMerchantCountryCode, getPaymentServiceChargeAmount, getPaymentTip, getPreferredCurrencyCode, getPrefTaxArray, getPrefTimeout, getUserExternalId, isInitialized, isNetworkConnected, preauthorizePayment, removePaymentItem, setSignatureBytes, usesDebug, getCustomerSupportNumber, getMerchantAddressCity, getMerchantAddressLine1, getMerchantAddressLine2, getMerchantAddressZipCode
1.2		Adam K.	Updated section 3.6.2. Added sections 3.6.8 and 3.6.9.
1.2		Adam K.	Added information about location of specific interfaces, classes and enums
1.2		Adam K.	Added section 4. covering Chip'n'PIN controller implementation guidelines

1.1.5		Joanna B.	Added new error codes: NoSMSGatewayError, ReverseRejected, RefundRejected, ItemsNotSet, CurrencyNotSet, ItemLessThanZero, TransactionTypeNotSet, SignatureNotSet, CardDataNotSet, TipLessThanZero, InternalServerError, EmailError, ParameterIsNull
1.1.5		Joanna B.	Added new methods to access information about new payment getCardHolderFirstName, getCardHolderLastName
1.1.5		Joanna B.	Added new Util methods for Merchant
1.1.4		Joanna B.	Added two statuses CANCELED and PREAUTHORIZED
1.1.3		Kacper S.	Added new method +(void) setPaymentCallbackURL(String) in AcceptSDK, which can be used for setting the callback URL for SUCCESS/FAILURE payment confirmation scenarios.
1.1.3		Kacper S.	Added new method +(String) getPaymentCallbackURL() in AcceptSDK, which can be used for getting currently set callback URL for SUCCESS / FAILURE payment confirmation scenarios.

Copyright

Printed in Germany / European Union
Version 1.4.9
Last updated: September 2016

Trademarks

2 SDK Overview

AcceptSDK for Android allows developers to quickly add mobile payments to their application using ready-to-use controllers for magstripe readers and Chip'n'PIN devices.

This SDK also provides a straightforward way to communicate with the Accept backend to use its functionalities, such as: creating payments, sending receipts, making refunds, and retrieving payment history.

2.1 Requirements

To use this SDK, Android version 2.1 or later is needed.

The target project needs to include the following permissions in its manifest:

```
<uses-permissionandroid:name="android.permission.INTERNET"/>
<uses-permissionandroid:name="android.permission.RECORD_AUDIO"/>
<uses-permissionandroid:name="android.permission.MODIFY_AUDIO_SETTINGS"/>
<uses-permissionandroid:name="android.permission.MOUNT_UNMOUNT_FILESYSTEMS"/>
<uses-permissionandroid:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permissionandroid:name="android.permission.ACCESS_NETWORK_STATE"/>
```

When used with Thyron Extension, SDK requires additional permissions, such as:

```
<uses-permissionandroid:name="android.permission.BLUETOOTH"/>
<uses-permissionandroid:name="android.permission.BLUETOOTH_ADMIN"/>
```

2.2 Setting up the SDK - Android Studio

Follow the steps described below to properly import the SDK project into Android Studio.(a. thyron extension, b. bbpos extension).

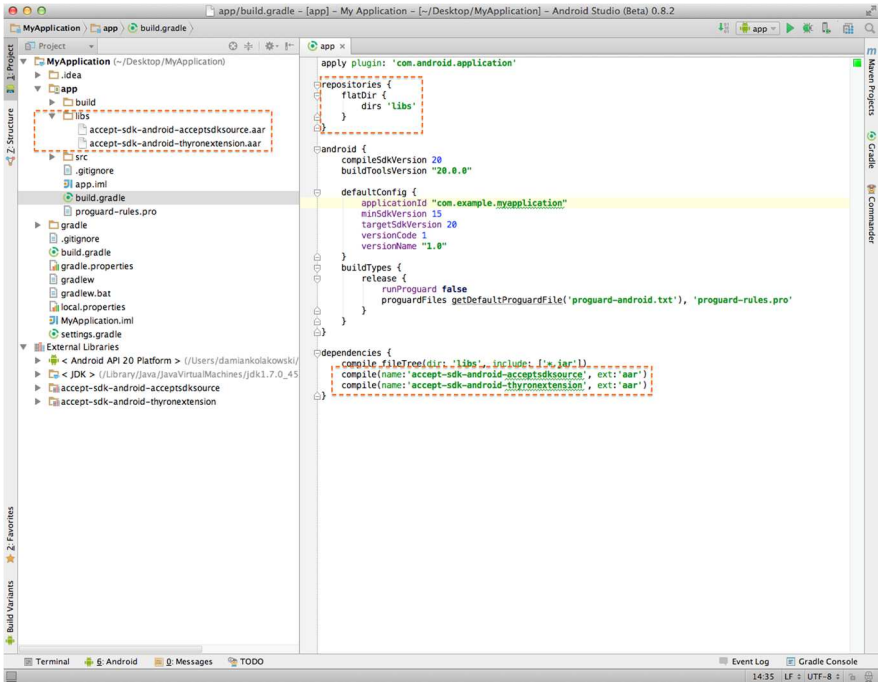
1. Depending on needed extension type support:
   a. Add: *accept-sdk-android-acceptsdksource.aar* and *accept-sdk-android-thyronextension.aar* files to your project (for example to libs directory) for support thyronextension.
   b. Add: *accept-sdk-android-acceptsdksource.aar* and *accept-sdk-android-bbposextension.aar* files to your project (for example to libs directory) for support bbposextension.

2. Modify application *build.gradle* file by adding *dependencies* with adding :

   a. compile(name:'accept-sdk-android-acceptsdksource', ext:'aar')
   compile(name:'accept-sdk-android-thyronextension', ext:'aar')
   b. compile(name:'accept-sdk-android-acceptsdksource', ext:'aar')
   compile(name:'accept-sdk-android- bbposextension, ext:'aar')

3. Modify application *build.gradle* file by adding *repositories*:
   a. flatDir { dirs 'libs' }

The following screenshot presents how the project should look after successfully importing the AAR files (please note, the *External Libraries* section will present a preview of AAR file content):



2.3 Setting up the SDK - Eclipse

Follow the steps described below to properly import the SDK project into Eclipse:

1. Unzip AAR files by changing files extensions from .AAR to .ZIP. The content of the AAR file should have the following structure:

   - /AndroidManifest.xml (mandatory)
   - /classes.jar (mandatory) - Accept SDK Sources.
   - /res/ (mandatory) - Accept SDK Resources.
   - /R.txt (mandatory)
   - /assets/ (optional)
   - /libs/*.jar (optional) - Additional Accept SDK dependencies.
   - /jni/<abi>/*.so (optional)
   - /proguard.txt (optional)
   - /lint.jar (optional)

2. Add extracted directories to Eclipse workspace and change their nature to Android Library Projects.
3. Add *classes.jar* files from Android Library Projects to *Build Path.*

2.4 Adding Thyron Extension to project

Currently AcceptSDK for Android includes an extensions system. Using this system, all implementations of magstripe readers and Chip'n'PIN controllers are now shipped as separate libraries. This also applies to Thyron's PosMate Smart terminal. Add the following configuration resources to the application to add a Thyron Extension to your project.

```
<!-- Instructs AcceptSDK to load Thyron Extension -->
<string-arrayname="extensions_list">
<item>ThyronExtension</item>
</string-array>

<!-- Adding this line will ensure that AcceptSDK uses PosMate terminal as
      CNPController -->
<item name="acceptsdk_terminal_type"type="string">thyron</item>

<!-- Adding this line will ensure that AcceptSDK uses Thyron terminal as
      Default extension -->
<item name="wl_default_terminal_type_use" type="string"
translatable="false">thyronExtension</item>

NOTE: SDK is using reflection to bind extension library to SDK
```

2.5 Adding BBPos Extension to project

Using this extensions system (described in 2.4), all implementations of magstripe readers and Chip'n'PIN controllers are supporting also separate library for BBPos's EMVSwipe terminal. Add the

For internal use only

following configuration resources to the application to add a BBPos Extension to your project.

```
<!-- Instructs AcceptSDK to load BBPos Extension -->
<string-array name="extensions_list">
    <item>bbposExtension</item>
</string-array>

<!-- Adding this line will ensure that AcceptSDK uses BBPos terminal as
      Default extension -->
<item name="wl_default_terminal_type_use" type="string"
translatable="false">bbPOSExtension</item>

NOTE: SDK is using reflection to bind extension library to SDK
```

3 SDK Usage

This section covers information about the structure of AcceptSDK for Android and its functions. It also contains guidelines on how to properly use this SDK.

3.1 Structure

The main class used to communicate with SDK is:

`de.wirecard.accept.sdk.AcceptSDK`

All models can be found in package:

`de.wirecard.accept.sdk.model`

All calls to SDK should be invoked directly from `AcceptSDK` class.
`AcceptSDK` also holds preferences and information about the merchant.

3.2 Models

The SDK provides ready-to-use models representing data exchanged between the mobile device and the backend.

3.2.1 PaymentItem

Represents a single item that takes part in a transaction. PaymentItem model implements the Parcelable interface.

Location: `de.wirecard.accept.sdk.model.PaymentItem`

Methods of this model are described in the following table:

Method	Description
- (int) getQuantity( )	Returns number of items sold
- (String) getNote( )	Returns the name or description of item
- (BigDecimal) getPrice( )	Returns gross price of single item
- (float) getTaxRate( )	Decimal in float format describing the tax rate of this item
- (BigDecimal) getTotalPrice( )	Returns amount times gross price
- (void) addItem( )	Increases amount by 1
- (void) removeItem( )	Decreases amount by 1
- (void) setQuantity(int amount)	Sets quantity to given amount
- (void) setPrice(BigDecimal value)	Sets price of single item to given value
- (void) setNote(String note)	Assigns new note

This example shows how to use the PaymentItem model:

```
BigDecimal price = new BigDecimal("12.5"),   //price of an item

LinkedList<PaymentItem> soldItems = new LinkedList<PaymentItem>();

PaymentItem singleItem = new PaymentItem (
      2,           //amount
      "<sold item name>",//item name
      price,       //BigDecimal object representing gross price of single item
      0.1f         //10% tax rate
);
singleItem = singleItem.addItem();     //Increase amount of items sold by 1
singleItem = singleItem.removeItem();  //Decrease amount of items sold by 1

soldItems.add(singleItem);
```

3.2.2 Payment

This model represents a single transaction stored in the backend. It is used in many requests in order to retrieve information or make changes in the backend. The payment model is immutable and implements the Parcelable interface.

Location: `de.wirecard.accept.sdk.model.Payment`

A description of its methods is described below:

Getter method	Description of return value
- (String) getTransactionId()	Is unique in the whole system and is linked with the transaction as a process, not with a single Payment object. Status of transaction can be found in status field.
- (TransactionType) getTransactionType()	Represents the type of payment.
- (Status) getStatus()	Enum; represents status of payment
- (Date) getProcessDate()	The date at which the payment was processed
- (BigDecimal) getTotalAmount()	Total value of this payment (sold items and optionally service charge at client expense or tip)
- (BigDecimal) getItemsTotalPrice()	Combined price of all sold items.
- (String) getCountryCode()	Country code of location where transaction was completed.
- (BigDecimal) getTip()	Tip for this transaction
- (BigDecimal) getServiceCharge()	Amount charged for this transaction
- (Location) getLocation()	Location of payment
- (String) getNote()	Description of payment

- (Integer) getTotalTax()	Value of tax in same currency as the items value
- (String) getSignature()	URL of signature image file
- (ArrayList<PaymentItem>) getPaymentItems()	List of items that took part in transaction
- (String) getCardHolderFirstName()	First name of card holder
- (String) getCardHolderLastName()	Last name of card holder
- (String) getCardNumber()	Last four digits of card number
- (float) getTipTaxRate()	Tip tax rate
- (String) getTerminalID()	Returns ID of terminal used during transaction. This is not the serial number of the device, but a number assigned by the acquirer.
- (String) getMerchantID()	Returns ID of merchant controlling the transaction. This ID is assigned by the acquirer.
- (String) getApplicationLabel()	Name of application used in transaction (e.g. VISA) retrieved from card's chip
- (String) getApplicationID()	ID of ICC application returned by the terminal
- (String) getTransactionCertificate()	Hash value of certificate signing the transaction, Provided by card when available.
- (String) getTVR()	Results of transaction verification performed by terminal.
- (String) getAuthorizationNumber()	Returned by acquirer in authorization process.
- (String) getTransactionStatusInformation()	Contains information about actions performed on terminal during transaction processing.
- (String) getIssuerAuthorizationCode()	Holds information about result of authorization performed by issuer. Although value '00' indicates successful authorization, transaction might be still rejected by terminal, card or acquirer during capture.

3.3 Enums

AcceptSDK class holds three enums: TransactionType, Status and GratuityType. A description of their usage and fields can be found in this section.

3.3.1 Transaction Type

Defines the method by which the client paid for this transaction.

Location: `de.wirecard.accept.sdk.AcceptSDK.TransactionType`

Possible values are:

TransactionType.CASH_PAYMENT - transaction was finalized using cash
TransactionType.CARD_PAYMENT - transaction was finalized using signature debit card or credit card

This enum is used during the creation of a new payment as well as in payments returned by the backend.

3.3.2 Status

This enum is contained in payment objects returned from the server. It holds information about status of payment.

Location: `de.wirecard.accept.sdk.AcceptSDK.Status`

Possible values are:

APPROVED - transaction has been finalized successfully
REJECTED - payment was rejected by server
REFUNDED - payment has been refunded to client
PREAUTHORIZED- the amount has been frozen on the account and transaction is ready to be captured
CANCELLED- transaction has been cancelled
PENDING- transaction is waiting for processing, can go to the REJECTED state if timeout, or to the APPROVED after successful transaction confirmation request

3.3.3 GratuityType

This enum is used to determine how service charge should be calculated when creating payments.

Location: `de.wirecard.accept.sdk.AcceptSDK.GratuityType`

It needs be passed to AcceptSDK.init() method during initialization to set default gratuity handling type. To change gratuity type after initialization, call AcceptSDK.setGratuityType(GratuityType) method. There are three options to choose:

GratuityType.NONE - payments don't support tips and service charge is covered by merchant,
GratuityType.TIP - payments support adding tips and service charge is covered by the merchant,
GratuityType.SERVICE_CHARGE - payments don't support tip, but service charge is covered by the customers.
GratuityType.TIP_AND_SERVICE_CHARGE – combination of previous two.

3.4 Initialization

Before making any calls to the SDK, it needs to be initialized. In order to do this, call `AcceptSDK.init()` method.

This example could be placed in `onCreate()`method of your main activity:

```
Context context = this;
String clientID ="<client id>";// id obtained from backend
String clientSecret = "<client secret>";// secret obtained from backend
String apiPath = "<url to backend api>";
String default = "<name of default extension>";//use static string in Extension
class

AcceptSDK.init(context, clientID, clientSecret, apiPath);
AcceptSDK.loadExtensions(context, default); // loading extensions libs using
reflection
AcceptSDK.setPrefTimeout(15); // sets the timeout (in seconds) for all requests
created by SDK
```

By default, the SDK has debug mode turned on. This results in printing logs for the purpose of easier debugging. If there is a need to change this behavior, the debug value can be changed by calling `AcceptSDK.setDebug(<false / true>)`method.

If there is a need to free resources held by the SDK, method `AcceptSDK.finish()` should be invoked. Keep in mind that after this call, all requests will fail until another `init()` is called.

Steps after success login:

- We have an async task for downloading the AID configuration (`AcceptSDK.getAcceptTerminalAIDConfiguration(getApplicationContext(), true); // true means downloadAndSaveToCache`)
- after successful download create CNP controller: CNPController cnpController = AcceptSDK.getCNPController();
- cnpController have information about the last connected device (controller.getLastConnectedDevice()). But for the first time it will be null value.
- cnpController provides a list of bounded CNP devices (cnpController.getBoundDevices()). We check the size of this list and if the list size equals 1 we got this device and load configurations to this device.

Our implementation:

```
CNPController cnpController = AcceptSDK.getCNPController();
if (cnpController == null)
        return;
CNPDevice cnpDevice = getLastConnectedDevice();

if (cnpDevice == null && cnpController.getBoundDevices().size() == 1)
        // load configuration to terminal
        cnpDevice = (CNPDevice) cnpController.getBoundDevices().get(0);

if (cnpDevice == null && cnpController.getBoundDevices().size() > 1)
        // Show a terminal chooser dialog
        cnpDevice = terminalChooserDialogValue();

if (cnpDevice == null)
        return;

cnpController.connectAndConfigure(cnpDevice, new CNPController.ConfigureListener()
{ ... });
```

*__Note__: This section should also cover the case if cnpController.getBoundDevices().size() > 1 *

- Like offering a selector for terminals (Android)
- Like checking the device configuration files (contactless.cfg=SPm2)

3.5 Creating new payment

To start the process of creating a new payment, call `AcceptSDK.startPayment()`. This method will return a unique id as a String to identify the payment during the process. Do not confuse this id with transaction id, as it is not related to the transaction in any way. To obtain the id of the payment that is currently being created, call `AcceptSDK.currentPayment()`method.

There are a number of parameters that need to be set in order to make a complete payment. The following example shows how to set these parameters:

```
AcceptSDK.startPayment();

// setting current location of merchant
// this parameter is optional
Location location = <current location obtained from GPS>;
AcceptSDK.setPaymentLocation(location);

// setting path to image file with signature of client
// It is required with card payments
String filePath="<path to image file with client signature>";
AcceptSDK.setPaymentSignature(filePath);

// setting list of items taking part in transaction
// only quantity, price and tax rate are required
ArrayList<PaymentItem> itemsList = new ArrayList<PaymentItem> ();
(...) // fill the list with items
AcceptSDK.setPaymentItems(itemsList);

// setting the type of transaction
// cash and card payments are processed differently, so It is important
// to make sure that this value is set properly
// this value is required
TransactionType paymentType = TransactionType.CARD_PAYMENT;
AcceptSDK.setPaymentTransactionType(paymentType);

// setting received tip as amount
// required when gratuity type is set to TIP
AcceptSDK.setPaymentTipAmount(<BigDecimal object>);

// setting tip tax rate
AcceptSDK.setPaymentTipTaxRate(<float>);

// setting received tip as percent of total price
// required when gratuity type is set to TIP
this tipPercent = 0.05f;
AcceptSDK.setPaymentTipPercent(tipPercent);
```

After setting these values, the payment is ready to be sent. In order to see how to accomplish this, refer to section 3.6.2.

In this example, the tip was set twice. Once as a fixed value (`setPaymentTipAmount`) and then as a percentage (`setPaymentTipPercent`). Only one kind of tip is taken into account when sending a new

payment to backend. The SDK chooses tip type based on the latest call used to set the tip, so in this example the 5% tip will be used (5% value is calculated from the total price of items sold).

To destroy resources held by the new payment, call `AcceptSDK.finishPayment()` method.

3.6 Communicating with backend

The SDK allows either retrieving data and making changes in backend. All calls are asynchronous and optionally take a listener as a parameter. This gives the possibility to handle the return value of those calls as well as the state with which they are returned.

It is important to remember that the listener is declared as a generic type class:

```
publicinterface OnRequestFinishedListener<T> {...}
```

Location: `de.wirecard.accept.sdk.OnRequestFinishedListener`

While making calls, make sure to pass the proper class type to the listener declaration, based on the type of request that is performed.

The following code shows an example declaration of this listener:

```
import de.wirecard.AcceptSDK.android.sdk.OnRequestFinishedListener;
(...)
private OnRequestFinishedListener<Object>loginListener =
	new OnRequestFinishedListener<Object>() {
	@Override
	publicvoid onRequestFinished(final ApiResult apiResult, final Object result)
	{
		if (apiResult.isSuccess()) {
			//Request succeeded, result object can be handled
		} else {
			//Request failed, handle error
		}
	}
};
```

A listener declared in this way can be passed to all requests, provided its generic type is consistent with the type of call's return value.

3.6.1 Merchant authentication

All calls to retrieve or change data on the server need to be authorized using a token. The client application can obtain this token by calling the `AcceptSDK.login()` method. A listener passed to request will receive a standard APIResultCode, but the returned object will be null. To get the token after authentication, method `AcceptSDK.getToken()` needs to be called. The token is stored between sessions in SharedPreferences. This means, that `login()` method does not need to be called on every application launch. The following is an example:

```
this.mToken = AcceptSDK.getToken();
if(mToken  ==  null)  {
	String userEmail = "foo@bar.com"; // e-mail address registered in backend
	String userPassword = "password";
	OnRequestFinishedListener mListener = new
OnRequestFinishedListener<Object>()    {
```

```
	@Override
		publicvoid onRequestFinished(ApiResult apiResult, Object result) {
			if(apiResult.isSuccess()) {
				MyClass.this.mToken = AcceptSDK.getToken();
			}
		}
	}
	AcceptSDK.login(userEmail, userPassword, mListener);
}
```

There can be one login request at any given moment. Any attempt to send a new login request, while there is already one pending, will result in the new request being reject (it will not be sent). In case there is a need to cancel the current login request, method `AcceptSDK.cancelLogin()` should be called.

In order to terminate the session, call `AcceptSDK.logout()` method.

3.6.2 Sending new payment

The SDK allows new payments to be sent to the backend or to finalize a preauthorized one.
A payment sent to the backend is registered in the system and if `apiResult.isSuccess()` returns true, the request triggers a real money transaction (only in production environment).
Payments are processed differently based on their transaction type.

In order to send a payment to the server, method `AcceptSDK.sendPayment()` needs to be invoked.

The following example could be bound to the payment button to finalize the transaction.

```
AcceptSDK.startPayment();
(...) // see section 3.5, to learn how to fill new payment with data

// optional, usually required when using chip and PIN controllers
AcceptSDK.preauthorizePayment(...);

(...)
OnRequestFinishedListener<Payment>paymentListener = new
OnRequestFinishedListener<Payment>() {
	@Override
	publicvoid onRequestFinished(final ApiResult apiResult, final Payment
result)	{
		if (apiResult.isSuccess()) {
			//payment succeeded, result object can be processed now
		} else {
			//payment failed with error
		}
	}
};
AcceptSDK.sendPayment(paymentListener);
```

Keep in mind that after calling this method, the newly created payment is still held by AcceptSDK. To avoid sending it again by mistake, you could call `AcceptSDK.finishPayment()` method.

The payment object returned by the request is not connected with any new payment stored in AcceptSDK.

It is not possible to cancel a currently pending payment request, but it is possible to check if the request is still running by calling `AcceptSDK.isSendPaymentRunning()` method.

3.6.3 Sending receipt

Receipts can be sent to clients either by SMS or e-mail. This can be accomplished by sending a request to the backend, which will handle receipt delivery. The following code shows how to create such a request sending a receipt to the customer's e-mail:

```
//You can retrieve this id from Payment object returned by backend
String uniqueId = "<uniqueId linked with transaction>";
String email = "<email address of receipt receiver>";
OnRequestFinishedListener<Object> receiptListener = new
OnRequestFinishedListener<Object>() {
	@Override
	publicvoid onRequestFinished(final ApiResult apiResult, final Object result)
	{
		if (apiResult.isSuccess()) {
			//receipt was successfully sent
		} else {
			//receipt wasn't sent
		}
	}
};
AcceptSDK.sendReceiptByEmail(uniqueId, email, receiptListener);
```

Sending the receipt by SMS is very similar:

```
String uniqueId = "<uniqueId linked with transaction>";
String phoneNumber = "<phone number of receipt receiver>"; // e.g. "+48123456789"
OnRequestFinishedListener<Object> receiptListener = new
OnRequestFinishedListener<Object>() {
	...
};
AcceptSDK.sendReceiptBySMS(uniqueId,phoneNumber,receiptListener);
```

It is possible to cancel this request by calling `AcceptSDK.cancelSendReceiptRequest()` method.

3.6.4 Getting list of payments

It is possible to receive a list of payments from the backend. The returned list is sorted by date (from newest to oldest). The following example shows how to send this request:

```
// getting payments 0-24
int pageNumber = 0;
int pageSize = 25;

// listener with proper generic type
OnRequestFinishedListener<List<Payment>> paymentsListener = new
OnRequestFinishedListener<List<Payment>>() {
	@Override
	publicvoid onRequestFinished(final ApiResult apiResult,
		final List<Payment> result)
	{
		if (apiResult.isSuccess()) {
			//payments list can be processed now
		} else {
			//error occurred while processing request
```

```
            }
      }
};

AcceptSDK.getPaymentsList(pageNumber, pageSize, paymentsListener);
```

It is possible to cancel this request by calling `AcceptSDK.cancelPaymentListRequest()` method.

3.6.5 Searching for payments

Searches can be conducted on the following information:
card holder first name,
card holder last name,
card number.

It is not possible to search by more than one field at once. In order to send a search request, use the following example:

```
// getting payments 0-24
int pageNumber = 0;
int pageSize = 25;

// query sent to backend
String query = "John";

// listener with proper generic type
OnRequestFinishedListener<List<Payment>> paymentsListener = new
OnRequestFinishedListener<List<Payment>>() {
      @Override
      publicvoid onRequestFinished(final ApiResult apiResult, final List<Payment>
result)      {
            if (apiResult.isSuccess()) {
                  //payments list can be processed now
            } else {
                  //error occurred while processing request
            }
      }
};
AcceptSDK.searchPayments(query, pageNumber, pageSize, paymentsListener);
```

The returned list is sorted by creation date from newest to oldest.

It is possible to cancel this request by calling `AcceptSDK.cancelSearchRequest()`method.

3.6.6 Refunding payment

It is possible to refund payments with finished status. In order to do this, `AcceptSDK.refundPayment()` method needs to be invoked. The following example shows how to accomplish this:

```
Payment payment = <Payment object retrieved in other request>;
String transactionID = payment.getTransactionId();
OnRequestFinishedListener<Payment> refundListener = new
OnRequestFinishedListener<Payment>() {
      @Override
```

```
	publicvoid onRequestFinished(final ApiResult apiResult, final Payment
result)	{
		if (apiResult.isSuccess()) {
			//payments has been refunded
		} else {
			//error occurred while processing request
		}
	}
};
AcceptSDK.refundPayment(transactionID, refundListener);
```

It is possible to cancel this request by calling `AcceptSDK.cancelReverseRequest()` method.

3.6.7 Getting single payment

In order to fetch information about a single payment, method `AcceptSDK.getPayment()` needs to be called. The following is an example showing how to use it:

```
String transactionID = "<ID of payment>";

OnRequestFinishedListener<Payment> paymentListener = new
OnRequestFinishedListener<Payment>() {
	@Override
	publicvoid onRequestFinished(final ApiResult apiResult, final Payment
result)	{
		if (apiResult.isSuccess()) {
			//payment object can be processed now
		} else {
			//error occurred while processing request
		}
	}
};
AcceptSDK.refundPayment(transactionID, refundListener);
```

It is possible to cancel this request by calling `AcceptSDK.cancelGetPaymentRequest()`method.

3.6.8 Preauthorizing payment

In contrast to the 1-step transaction creation explained in section 3.6.2., the AcceptSDK also provides the ability to create 2-step transactions. Preauthorization is the first step, while capture is the second. Preauthorizing a payment does not subtract the amount specified in the transaction from the client's account. Instead, it secures this amount, preventing it from being used at the time of processing the payment by any other payment system or transaction.
If the capture is not performed within 15 minutes from successful preauthorization, the secured amount is made available for any processing again, and the transaction process must be restarted.

To preauthorize the payment, follow this example:

```
OnRequestFinishedListener<PreauthResult>preauthListener = new
OnRequestFinishedListener<PreauthResult>() {
	@Override
	publicvoid onRequestFinished(final ApiResult apiResult, final PreauthResult
result)	{
		if (apiResult.isSuccess()) {
			//preauthorization succeeded, result object can be processed
now
```

```
            } else {
                  //preauthorization failed.
            }
      }
};
AcceptSDK.preauthorizePayment(preauthListener);
```

Once again, successful preauthorization does not indicate transaction completion. Its purpose is to check the validity of the provided data and the availability of the requested funds. Successful preauthorization does NOT indicate that in the end the transaction will be approved.

This request can be cancelled by calling `AcceptSDK.cancelPreAuth()` method.

3.6.9 Multi-currency

Since version 1.3.4, the SDK exposes the API to load a list of currencies defined on the backend side. The following method is used to obtain the list:

```
final Set<String> allowedCurrencies = AcceptSDK.getAllowedCurrencies();
```

Currency used for further transactions can be changed using following method:

```
AcceptSDK.setDisplayCurrency("EUR");
```

3.6.10 Capturing the payment

After preauthorization and optional additional processing, payment can be captured. To capture the payment using AcceptSDK, call `AcceptSDK.sendPayment()` method. See section 3.6.2. for more information.

3.6.11 Terminal configuration

Terminal configuration can be obtained using AcceptSDK.fetchTerminalConfiguration(..) method. The following examples shows how to use the method and how a proper error handling should look.

```
final Response<AcceptTerminalConfiguration> response =
AcceptSDK.fetchTerminalConfiguration(...serial..number...);
if ( response.hasError() ) {
    switch ( response.getError().type ) {
        case ErrorType.IO: /* inform user about connection error */ return;
        case ErrorType.DESERIALIZER: /* no JSON in response or body is not a
valid JSON data or model of response is invalid. Details available as
response.getError().toString()  */ return;
        case ErrorType.HTTP:
        {
            switch ( response.getError().httpErrorCode ) {
                case 422: /* Terminal configuration is not ready. Accept
backend needs to be configured to serve terminal configuration for given
serial number. */ return;
                default:
                    /* Report unknown error with content of
response.getError().toString() */
                return;
        }
```

```
        default:
        {
            /* Report unknown error with content of
response.getError().toString() */
            return;
        }
    }
}
final AcceptTerminalConfiguration configuration = response.getBody();
```

3.7 Request errors handling

ApiResult object holds feedback information from the backend request.

Location: `de.wirecard.accept.sdk.ApiResult`

Using `result.isSuccess()` is the most straightforward way to determine if the request was successful.

```
ApiResult result;

// Most important parameter. Points if the request was successful.

boolean success = result.isSuccess();
```

More specific information about the request can be found in enum Code. It holds information about the status of the request.

```
// Holds information about the status of the request
Code code = result.getCode();
```

All possible values are:

Ok	Request was successful.
Cancelled	Request was cancelled.
Network	Signals network problems.
IllegalArguments	Incorrect parameters were provided.
ServerDeploy	Server might be unavailable or under deployment.
Unauthorized	Authentication failed. Resources are denied by the server. Token for session has not been given or has expired.
ValidationError	Validation error. Some fields may contain incorrect data.
SMSError	Error sending SMS request. Phone number may be incorrect.
MerchantDisabled	Merchant is disabled and cannot perform transactions.
MerchantLimitExceeded	Limit for the merchant has been exceeded.
WirecardError	Error that occurred making request to the Wirecard gateway.
Unexpected	Unknown or unexpected error.
NoSMSGatewayError	Error sending SMS receipt. This function is unsupported.
ReverseRejected	Reverse was rejected.
RefundRejected	Refund was rejected.
ItemsNotSet	Payment items are not set.

CurrencyNotSet	Currency for the payment is not set.
ItemLessThanZero	Payment item price must be greater than zero.
TransactionTypeNotSet	Transaction type is required for payments.
SignatureNotSet	Signature is required for signature credit card payments.
CardDataNotSet	Card data was not set. Swipe card first.
TipLessThanZero	Payment tip should not be less than zero.
InternalServerError	Internal server error.
EmailError	Error sending e-mail receipt. May be due to illegal e-mail provided.
ParameterIsNull	One of the passed parameters is null
PaymentMethodNotSet	Method by which payment should be processed is not set

A detailed description of the error that has occurred can be obtained using `result.getMessage()` :

```
// Additional information / hint. Might signal reasons why request has failed.

// This element might hold multiple advises if one back-end call required multiple
http requests.

String message = result.getMessage();
```

3.8 Helper methods

AcceptSDK provides various helper methods. All these methods should be called directly from AcceptSDK class. A list can be found in this section.

3.8.1 Getting merchant preferences

```
// Getting token of current section (null if no section was created):
+(String) getToken();

// Getting minimum amount of money needed to accept the payment.
+(BigDecimal) getPrefMinAmount();

// Getting maximum amount of money needed to accept the payment
+(BigDecimal) getPrefMaxAmount();

// Getting ISO 4217 string representing currency set in merchant account
+(String) getPreferredCurrency();

// Numeric representation of current currency code
+(long) getPreferredCurrencyCode();

// If ‘true’ is returned, tax values are being processed. If value is set to
‘false’, all tax values in PaymentItem objects are processed as 0 value (even if
different value is set)
+(boolean) getPrefTaxationEnabled();

// If ‘true’ is passed to this method, taxation is enabled. This value persists
between sessions. By default, taxation is enabled.
+(void) setPrefTaxationEnabled(boolean enabled);
```

3.8.2 Methods that can be used when creating new transaction

```
// Getting string representation of new payment:
+(String) printPayment();

// Getting combined price of payment items from new payment:
+(BigDecimal) getPaymentAmount();

// Getting total price summed with tip and service charge:
+(BigDecimal) getPaymentTotalAmount();

// Getting location of new payment:
+(Location) getPaymentLocation();

// Getting transaction type of new payment:
+(TransactionType) getPaymentTransactionType();

// Getting path to the image file with client signature of new payment:
+(String) getPaymentSignature();

// Getting list of items of new payment:
+(List<PaymentItem>) getPaymentItems();

// Removing specified payment item from transaction:
+(void) removePaymentItem(PaymentItem item);
```

```
// Getting path to the image file with client signature of new payment:
+(String) getPaymentSignature();

// Getting the card holder name from most recent swipe assigned to new payment:
+(String) getCurrentCardHolderName();

// Getting first names of card holder (if available)
+(String) getCardHolderFirstName();

// Getting last names of card holder (if available)
+(String) getCardHolderLastName();


// Getting the card number from most recent swipe assigned to new payment:
+(String) getCurrentCardNumber();

// Getting currently set callback url for payments
+(String) getPaymentCallbackURL();

// Getting the tip tax rate for new payment
+(Float) getPaymentTipTaxRate();

// Getting currently set gratuity type
+(GratuityType) getGratuityType();

// Getting currently set currency
+(String) getPaymentCurrency();

// Getting tip amount
+(BigDecimal) getPaymentTip();

// Getting service charge amount
+(BigDecimal) getPaymentServiceChargeAmount();

// Getting the array of tax rates
+(ArrayList<String>) getPrefTaxArray()

// Removes signature bytes stored in SDK
+(void) clearSignature()
```

3.8.3 Util

```
// Getting date format consistent with backend:
+(SimpleDateFormat) getDateFormat();

// Getting information if debug is enabled
+(boolean) usesDebug()
// Returns true if AcceptSDK is initialized
+(boolean) isInitialized()

// Returns true if device has internet access
+(boolean) isNetworkConnected()

// Returns time in seconds after which connection times out when there is no
transmission
```

```
+(int) getPrefTimeout()

// Returns Set of available magstripe readers’ names.
+(Set<String>) getSupportedSwipers()

// Loads new implementation of magstripe reader. Parameter passed must be one of
the names retrieved using getSupportedSwipers() method.
+(void) loadSwiperByName(String name)

// Returns Set of available CNP controllers’ names.
+(Set<String>) getSupportedCNPControllers()

// Loads new implementation of CNP controller reader. Parameter passed must be one
of the names retrieved using getSupportedCNPControllers() method.
+(void) loadCNPControllerByName(String name)
```

3.8.4 Merchant Util

```
// Getting default locale set for merchant
+(String) getPreferredLocale();

// Getting merchant name
+(String) getMerchantName();

// Getting merchant contact phone number
+(String) getMerchantPhoneNumber

// Getting merchant contact email address
+(String) getMerchantEmail();

// Getting merchant external id (external id is custom username; It is optional)
+(String) getUserExternalId();

// Getting username (used during login)
+(String) getUserName();

// Getting merchant country code
+(String) getMercantCountryCode();

// Getting external id used to login
+(String) getUserExternalId();

// Getting customer support phone number.
+(String) getCustomerSupportNumber();

// Getting name of city in which merchant resides
+(String) getMerchantAddressCity();

// Getting first line of merchant’s address
+(String) getMerchantAddressLine1();

// Getting second line of merchant’s address
+(String) getMerchantAddressLine2();

// Getting zip-code assigned to merchant’s address
+(String) getMerchantAddressZipCode();
```

3.9 Sending Logs

Since AcceptSDK is still in development phase and new features are constantly being added, it may occur that the development team implementing the SDK encounters a problem which it is unable to solve on their own. In that case, we advise using our log sending feature. Simply call `AcceptSDK.sendDebugMail()` method (for example after the transaction has been declined), providing Context and List of e-mail addresses as parameters. This will launch e-mail sending intent, with all necessary fields filled. Confidential data, such as transaction ID, signatures, etc. is masked to protect privacy. Sensitive data such as passwords is not contained in these logs. Please contact us if you have troubles choosing the recipients of log messages, so that we are able to assign someone to your issue.

3.10 Session handling

For security reasons, sessions started in login process have an expiration time set. When the session expires, the access token used by the SDK is no longer valid and the attendant should log in again in order to continue using the provided functionality. Currently, there are two ways in which the SDK indicates session expiration:

`Code.Unauthorized` being returned in error response,
registering *BroacastReceiver* using *IntentFilter* with *`AcceptSDKIntents.SESSION_TERMINATED`* action. This Intent is broadcasted even when no request was sent.

By default, merchant accounts have a 15-minute expiration timer, but this may vary. The session is refreshed every time the request requiring authentication is successfully sent to the backend.

3.11 Receipt

There is a ReceiptBuilder.java class for providing the data from payment to build the receipt.

Location: `de.wirecard.accept.sdk.util.ReceiptBuilder`

Methods of ReceiptBuilder are described in the following table:

Method	Description
- (void) writeToSDFile(String)	Create receipt html file in **/accept** folder with String content
- (String) getMerchantNameAndSurname()	Return name and surname of Merchant
- (String) getMerchantAddressLine1()	Return address line 1 of Merchant
- (String) getMerchantAddressLine2 ()	Return address line 2 of Merchant
- (String) getMerchantAddressCity()	Return city address of Merchant
- (String) getMerchantAddressZipCode()	Return address zip code of Merchant
- (String) getMerchantCountryCode()	Return country code of Merchant
- (Bigdecimal) getTransactionAmount(Payment)	Return total transaction amount for Payment
- (String) getTransactionCurrency(Payment)	Return transaction currency for Payment

- (long) getTransactionDate(Payment)	Return transaction process date for Payment
- (String) getTransactionDate(Payment, SimpleDateFormat)	Return transaction process date for Payment in SimpleDateFormat
- (List<String>) getDescriptionOfGoodsAndServices (Context, Payment)	Return list of descriptions of all PaymentItems in the Payment
- (String) getCardholderSignature (Payment)	Return link to cardholder signature for Payment if exists
- (String) getTransactionType (Payment)	Return EMV transaction type for Payment
- (String) getCardNumber (Payment)	Return card number for Payment
- (String) getAuthorizationCode (Payment)	Return payment authorization code
- (String) getAID (Payment)	Return application ID for Payment
- (String) getApplicationLabel (Payment)	Return application label tag value from TLV data if data is available
- (String) getReceiptNumber (Payment)	Return receipt number of Payment
- (List<PaymentItems>) getTransactionItems (Payment)	Return list of Payment Items from Payment
- (AcceptSDK.Status) getTransactionStatus (Payment)	Return transaction status for Payment
- (String) getCardHolderFirstName (Payment)	Return cardholder first name
- (String) getCardHolderLastName (Payment)	Return cardholder last name
- (String) getTerminalID (Payment)	Return terminal ID
- (String) getMerchantID (Payment)	Return merchant ID
- (Bigdecimal) getTransactionTip (Payment)	Return amount of Payment tip
- (Bigdecimal) getTransactionServiceCharge (Payment)	Return amount of Payment service charge
- (boolean) isServiceChargeAdded (Payment)	Return true if service charge was added to Payment

4 Chip&pin and Swiper devices support

This section describes how to incorporate support for chip&pin and swiper devices using AcceptSDK.

4.1 Introduction

Currently AcceptSDK has support for Chip'n'PIN and Swiper devices. This allows processing transactions using ICC cards. The SDK itself contains no implementation for those devices, but it has a basic interface which can be used to add one. Usually the implementation is shipped as a separate library in the form of an extension, and any requirements specific for the given device are described in its documentation. This document describes the generic CNPController interface (for C&P read Chip&pin) and Swiper interface. It should be used as a guideline on implementing the controller and swiper and an introduction to the related structures.

4.2 Using mock configuration for Chip&pin terminal

Performing transactions using Chip'n'PIN controllers requires proper setup. AcceptSDK already takes care of this, but a merchant user used in development might not always have the required data set on his account. To prevent this issue from stopping the dev team from making progress, AcceptSDK comes with mock EMV data which might be enabled on demand. To do this, add the following line to one of your project's resource xml files:

```
<itemname="acceptsdk_mock_cnp_values"type="bool">true</item>
```

After building the application, be sure to log out and log in again. After this, the mock data should be in use. It is important to remember to turn this feature off when your merchant user has configuration data set on account.

It should also be noted that this configuration may be different to the one which will be used in the production environment. Because of this, results received while processing transactions using mock data may differ from the ones in live setting.

4.2.1 Mock configuration profiles

Currently there are 2 predefined profiles present in SDK, named Profile 1 and Profile 2. While Profile 1 forces transactions to be approved online, Profile 2 allows authorization to be performed completely offline. Such transactions still require an internet connection to be finalized. The following tables contain configuration values present in both profiles:

Description	Value
Merchant name	Mock profile 1
POS Entry Mode	Customer present, ICC, PIN
Terminal Type	21, Online only
Terminal Capabilities	All card verification methods available
Currency	Euro
Country	Germany
Floor limit	0 eurocents
Should transactions be forced online	yes
TAC Default	DC4000A800

TAC Online	0010000000
TAC Denial	DC4000F800

Values of mock Profile 1 configuration

Description	Value
Merchant name	Mock profile 2
POS Entry Mode	Customer present, ICC, PIN
Terminal Type	22, Offline with online capability
Terminal Capabilities	All card verification methods available
Currency	Euro
Country	Germany
Floor limit	5000 eurocents
Threshold value	2500 eurocents
Should transactions be forced online	no
TAC Default	DC4000A800
TAC Online	0000000000
TAC Denial	DC4000F800

Values of mock Profile 2 configuration

Profile 1 is used by default. In order to use Profile 2, add the following entry to one of the resource xml files:

```
<itemname="acceptsdk_mock_cnp_selected_profile"type="string">@string/acceptsdk_moc
k_cnp_profile2</item>
```

Respectively, to ensure that Profile 1 is being loaded, value

`@string/acceptsdk_mock_cnp_profile1`

could be used.

4.3 Models

The following sections contain descriptions of models used in connection with the CNPController and Swiper by the controller itself. Most of them are generic, abstract types and have all methods required for implementation of the payment system. Implementations of those models are proprietary for each controller. Performing transactions should not require handling of more methods than there are present in the base interfaces.

4.3.1 CNPController

CNPController is the base class for all Chip'n'PIN controllers. Its instance can be accessed by calling `AcceptSDK.getCNPController()` method (if no controller is available, this method will return null).

Location: `de.wirecard.accept.sdk.cnp.CNPController`

The following table shows the list of this model's methods and their descriptions:

Method	Description
- (void) searchForDevices(DiscoveryListener listener)	Starts searching for devices. Found devices are reported to provided listener.
- (void) stopSearchingForDevices( )	Cancels device discovery.
- (void) disconnect( )	Terminates current connection.
- (float) voidTransaction( )	Cancels processing of the current transaction.
- (List<CNPDevice<?,?>>) getBounDevices( )	Returns the list of known devices.
- (void) destroy( )	Frees all resources held by controller
- (void) enableAdapter( )	Enables communication adapter (if applicable)
- (void) disableAdapter( )	Disables communication adapter (if applicable)
- (void) restartDevice( )	Orders device to restart (only if device is capable)
- (void) setPreauthorizationResult(PreauthResult result)	Provides preauthorization result to controller, allowing c'n'p device to validate data returned.
-(CNPDevice<?,?>) getLastConnectedDevice( )	Returns most recent CNPDevice with which connection was established. Returns null if no device was connected yet.
-(void) connectToDevice(CNPDevice<?,?> device)	Connects to specified device.
-(String) getDeviceType()	Returns the unique name for this CNPController implementation
-(void) startTransactionProcess(CardEntryType entryType, boolean allowGratuity), AcceptTerminalConfiguration configuration)	Order controller to start processing transaction. Confiuration parameter can be obtained from AcceptSDK.fetchTerminalConfiguration() method.
-(void) setCNPListener(CNPListener listener)	Listener provided in this method will receive events from controller.

An example usage of the above methods can be found in following sections.

As a reminder, CNPController should not be instantiated directly, using its constructor. Instead, call `AcceptSDK.getCNPController()` method.

4.3.2 CNPDevice

The CNPDevice model is a base model representing data present in common device types. It implements Parcelable interface. CNPDevice can be passed to `CNPController.connectToDevice()` method in order to begin the connection process.

Location: `de.wirecard.accept.sdk.cnp.CNPDevice`

Following table shows the list of this model's methods and their descriptions:

Method	Description
- (String) getName( )	Returns human-readable name (if available)
- (String) getAddress( )	Returns the unique address used to identify this device
- (boolean) isBound( )	Returns true if this device is bound / known
- (JSONObject) toJSON( )	Serialization of this object to JSON. This device can be then deserialized by using CNPDevice(JSONObject json) constructor.

To see how to obtain CNPDevice reference, check 'events handling and communication' section.

4.3.3 Swiper

Swiper is the base class for all Swiper controllers. Swipers are communicating with smartfone using audio output/input (jack). Its instance can be accessed by calling `AcceptSDK.getSwiper()` method (if no controller is available, this method will return null).

Location: `de.wirecard.accept.sdk.swiper.Swiper`

The following table shows the list of this model's methods and their descriptions:

Method	Description
- (void) register(OnSwipeDetectedListener listener)	Registration of listener. All events will be reported to provided listener.
- (void) unregister( )	Unregister listener.
- (void) init ( )	Initialize device controller defined in the official BBPos support library. Starts audio recording.
- (void) release ( )	Relese device controller,stop audio recording.
- (void) destroy( )	Frees all resources held by controller
- (boolean) isHeadsetConnected( )	Provide info about hardware connectivity
- (String) getSwiperInfo()	Provide basic info like “name/unique id”
- (void) sendOnlineprocessResult(String result)	Used to provide GoOnlineRequest results back to the swiper terminal
- (void) selectAid(int index);	Used to inform swiper about AID selection (terminal is without display)
- (void) finalConfirm(boolean result)	Used to provide final confirm info to terminal

An example usage of the above methods can be found in following sections.

As a reminder, Swiper should not be instantiated directly, using its constructor. Instead, call `AcceptSDK.getSwiper()` method.

4.3.4 OnSwipeDetectedListener

Listener handling events Swiper state changes.

Location: `de.wirecard.accept.sdk.swiper.OnSwipeDetectedListener`

SwipeEvent enum All possible values are:

CHARGING_UP	First request, starting swiper…waithing for first answer.
DEVICE_INFO	Received info message. TAG = DeviceInfo.class object
MESSAGE	Message. If TAG not null please check instance of (DisplayText.class, EMVError.class, TransactionResult.class)
AUTO_CONFIG	Auto configuration started.
DISCONNECTED	Terminal was disconnected.
UNRECOGNIZED_SWIPER	Swiper was not recognized.
CONNECTED	Terminal is connected.
ERROR	Describes error type. TAG = EMVError.class
FAILED_SWIPE	Failed swipe state, pls reswipe or insert the card.
EMV_STARTED	EMV transaction with chip started.
READY	Swiper is waithing for swipe or insert of chipcard.
READY_FOR_INSERT	Swiper is waithing for only for insert of chipcard.(probably bad swipe or broken Magstripe)
READY_FOR_SWIPE	Swiper is waithing for swipe. (Bad chip or wrong insert, fallback)
REQUEST_ONLINE_PROCESSING	Swiper request sendin data to the Issuer.
SELECT_AID	Show AID selector. TAG = (java.util.List<String>) aidNames. Depends on position.
FINAL_CONFIRM	Final conform requested.
TRANSACTION_RESULT	Transaction result response from terminal. TAG = de.wirecard.accept.sdk.swiper.TransactionResult.class

Method	Description
- (void) onSwipeSuccessful()	Event used to inform abou successful swipe. Continue on sending magstripe data online.
- (void) onSwiperEvent (SwiperEvent state, Object tag, String details)	Main event callback, hannling of all other situations. Tag Object type depends on situation, Details string is a describing message.

4.4 Enums

This section describes enums used in communication with CNPController. Some of the enums also contain action codes assigned to them giving more detailed information on why this value is provided. Action codes are described along with certain enum values.

4.4.1 CardEntryType

Specifies the way data should be read from the card. It should be provided to `CNPController.startTransactionProcess()` method.

Location: `de.wirecard.accept.sdk.cnp.CardEntryType`

Possible values are:

SWIPE - Terminal is forced to ask for magnetic stripe readout
ICC - Terminal is forced to ask for ICC card entry
SWIPE_OR_ICC - Terminal is free to ask for either card entry or card swipe based on its own set of rules.

4.4.2 ProcessState

This enum provides information about the current state of transaction processing. It is delivered to `CNPListener` object by `onProcessUpdate()` method.

Location: `de.wirecard.accept.sdk.cnp.observer.ProcessState`

Possible values are:
SWIPE_OR_INSERT - Indicates that terminal has prompted for card entry or swipe. This value has the following action codes, which can be accessed by calling `getActionCode()` method, providing more precise information:
ACTION_PROMPT_SWIPE - Terminal is waiting for card swipe,
ACTION_PROMPT_ICC - Terminal is waiting for ICC card entry,
ACTION_PROMPT_ICC_OR_SWIPE - Terminal is waiting for either of above (swiping the card might lead to prompt for ICC entry. In that case same, generic action will be provided though).
SIGNATURE_REQUIRED - Signature is required to verify this cardholder. For specific instructions on how to handle signature requests, refer to section 3.6.6. When using PosMate terminal, it needs to be passed to the attendant. At this point he has to press the green OK button and follow the instructions shown on the display. When the attendant approves the signature on the terminal, the `CNPListener.onProcessFinished()` method is called with a result based on the decision taken by the merchant:
SUCCESS - if signature was approved on terminal. At this point the signature provided by the client should be passed to the SDK by calling either `AcceptSDK.setSignatureBytes()` or `AcceptSDK.setPaymentSignature()` method. It is expected that soon after this event, the *CAPTURE_REQUIRED* state will be passed to this listener.
VOID - signature was rejected by the attendant on the terminal.
TERMINATING- *Transaction is being terminated either on the merchant's or the terminal's demand. It may take few seconds to finish.*
CARD_DATA_READ- Data was successfully read from the card after swipe or has been entered for Chip'n'PIN transaction.
RESTARTING - Terminal is beginning its restart procedure and the connection is expected to be terminated
PREAUTHORIZATION_REQUIRED- Preauthorization needs to be performed in order to continue with the transaction. To do this, call `AcceptSDK.preauthorizePayment()` method. PreauthResult returned in response should be then passed to CNPController's `setPreauthorizationResult()` method. This will order the terminal to validate the provided data and continue processing the transaction.
CAPTURE_REQUIRED- Capture needs to be performed. To perform a capture, call `AcceptSDK.sendPayment()` method. This is the last step in transaction processing. A successful capture should return Payment object containing the transaction data. There is no need to pass any more data to the terminal, as at this point the transaction processed by the terminal is finished. If a capture is not performed within 15 minutes from successful preauthorization, the transaction will be voided by the payment system and it will no longer be valid.
PROMPT_REMOVE_CARD- Terminal has prompted for ICC card removal.

4.4.3 ProcessResult

This enum is provided in `CNPListener.onProcessFinished()` method. It indicates that transaction processing finished with a result specified by one of the values below.

Location: `de.wirecard.accept.sdk.cnp.observer.ProcessResult`

Possible values are:
SUCCESS- Transaction processing on terminal has finished successfully. No more processing needs to be done on it. It is expected to receive a transaction status update with *CAPTURE_REQUIRED* value.
VOID- Transaction has failed and is now being voided on terminal. Processing this transaction cannot continue, and a new transaction needs to be started in order to try again.

4.4.4 AdapterEvent

This enum is provided in `CNPListener.onAdapterEvent()` method. It gives information about the state of transport layer specific for the given device.

Location: `de.wirecard.accept.sdk.cnp.observer.AdapterEvent`

Possible values are:
ADAPTER_DISABLED - *adapter is in disabled state and no communication link can be established.*
ADAPTER_ENABLING - *adapter is being enabled (going from* DISABLED *to* IDLE *state)*

- *ADAPTER_IDLE* - adapter is enabled and communication link can be established now. This value can have one of the following action codes assigned (can be accessed by calling *getActionCode()* method):
  - *ACTION_IDLE* - adapter entered *IDLE* state from *DISABLED* state
  - *ACTION_IDLE_CONNECTION_FAILED* - adapter entered *IDLE* state after connection attempt failed or was terminated
  - *ACTION_IDLE_CONNECTION_LOST* - adapter entered *IDLE* state because of problems on communication layer (for example timeout after successfully establishing connection).
  - *ACTION_IDLE_CONNECTION_FINISHED* - adapter entered *IDLE* state after disconnecting from c'n'p device on user's demand

ACTION_IDLE_CONF_UPDATE_FAILED - *adapter entered* IDLE *state because there were troubles when updating either firmware or configuration.*
ADAPTER_DISABLING - *adapter is being disabled (going from* IDLE *to* DISABLED *state) either on user's demand or because of internal failure.*

4.4.5 PreauthResult

This enum is returned as a result of a preauthorization request. It contains instructions for the terminal on how to continue with processing the transaction.

Location: `de.wirecard.accept.sdk.cnp.PreauthResult`

Possible values are:
SUCCESS - *request succeeded. It does not mean that the transaction will be approved, as instructions contained in issue response code and data may lead to the transaction being voided. Only after passing PreauthResult to* *CNPController.setPreauthorizationResult()* *method can the final verdict be known.*

- *DECLINED* - preauthorization was declined at early stage.
- *REFERRAL_NEEDED* - RFU

- *SIGNATURE_REQUIRED* - signature must be placed before capturing the payment. Most likely, after passing this PreauthResult object to CNPController, `CNPListener.onProcessUpdate()` method will be called with `SIGNATURE_REQUIRED` *state*.

CANCEL - *request was canceled on user demand.*

4.4.6 TerminalEvent

TerminalEvent enum provides information about asynchronous events, the occurrence of which is determined by proprietary implementation of the controller.

Location: `de.wirecard.accept.sdk.cnp.TerminalEvent`

4.4.7 DisplayText

DisplayText is Enum used to simply display some messages requested from terminal side.
Items are selfdescribed, defined by the bbpos in swiper.

Location: `de.wirecard.accept.sdk.swiper.DisplayText`

```
AMOUNT, AMOUNT_OK_OR_NOT, APPROVED, CALL_YOUR_BANK, CANCEL_OR_ENTER,
CARD_ERROR, DECLINED, ENTER_AMOUNT, ENTER_PIN, INCORRECT_PIN, INSERT_CARD,
NOT_ACCEPTED, PIN_OK,PLEASE_WAIT, PROCESSING_ERROR, REMOVE_CARD,
USE_CHIP_READER, USE_MAG_STRIPE, TRY_AGAIN, REFER_TO_YOUR_PAYMENT_DEVICE,
TRANSACTION_TERMINATED, TRY_ANOTHER_INTERFACE, ONLINE_REQUIRED, PROCESSING,
WELCOME, PRESENT_ONLY_ONE_CARD, CAPK_LOADING_FAILED, LAST_PIN_TRY,
INSERT_OR_TAP_CARD, SELECT_ACCOUNT, APPROVED_PLEASE_SIGN, TAP_CARD_AGAIN,
AUTHORISING, INSERT_OR_SWIPE_CARD_OR_TAP_ANOTHER_CARD,
INSERT_OR_SWIPE_CARD, MULTIPLE_CARDS_DETECTED
```

4.4.8 EmvError

EMVError is Enum used to define error types in swiper
Items are selfdescribed, defined by the bbpos.

Location: `de.wirecard.accept.sdk.swiper.EMVError`

```
UNKNOWN, CMD_NOT_AVAILABLE, TIMEOUT, DEVICE_RESET, DEVICE_BUSY,
INPUT_OUT_OF_RANGE, INPUT_INVALID_FORMAT, INPUT_ZERO_VALUES, INPUT_INVALID,
CASHBACK_NOT_SUPPORTED, CRC_ERROR, COMM_ERROR,
TERMINAL_CONFIGURATION_CONNECTION_ERROR,  TERMINAL_CONFIGURATION_NOT_FOUND,
CANT_LOAD_TERMINAL_SERIAL_NO, CARD_DATA_PROCESSING, LOW_BATTERY,
FAIL_TO_SEND_TC, DEVICE_INFO_LOAD, FAIL_TO_START_AUDIO,
FAIL_TO_START_BLUETOOTH, INVALID_FUNCTION_IN_CURRENT_MODE,
COMM_LINK_UNINITIALIZED, BTV2_ALREADY_STARTED,
AUTO_CONFIG_NOT_SUPPORTED_DEVICE, AUTO_CONFIG_INTERUPTED,
VOLUME_WARNING_NOT_ACCEPTED
```

4.4.9 TransactionResult

TransactionResult is Enum used to define possible results of transaction in swiper

Items are selfdescribed, defined by the bbpos.

Location: de.wirecard.accept.sdk.swiper.TransactionResult

```
APPROVED, TERMINATED, DECLINED, CANCEL, CAPK_FAIL, NOT_ICC, CARD_BLOCKED,
DEVICE_ERROR, CARD_NOT_SUPPORTED, MISSING_MANDATORY_DATA, NO_EMV_APPS,
INVALID_ICC_DATA, CONDITION_NOT_SATISFIED, APPLICATION_BLOCKED, UNKNOWN,
SELECT_APP_FAIL, ICC_CARD_REMOVED;
```

4.5 Events handling and communication with Chip&Pin

There are two interfaces that are usually required to properly implement the controller:
- DiscoveryListener, which allows searching for devices that might be used for transaction processing,
- CNPListener, which should be used for controlling the flow of the transaction and to react to specific events.

Their detailed description can be found in subsections 4.5.1. and 4.5.2.

Different implementations of CNPController should not require implementation of more listeners than those presented here. It is safe to assume that implementing them is enough to handle payments with different controllers.

4.5.1 Device discovery (DiscoveryListener)

CNPController allows searching for devices of a type specific for its implementation. To do this, simply call `CNPController.searchForDevices()` method, providing an instance of DiscoveryListener object.

Location: de.wirecard.accept.sdk.cnp.DiscoveryListener

To cancel a search, use `CNPController.stopSearchingForDevices()` method. It is also possible to retrieve a list of known devices by calling `CNPController.getBoundDevices()` method. The following code shows example usage of these methods:

```
// get instance of current CNPController
CNPController<?>controller = AcceptSDK.getCNPController();

// orders controller to start searching for devices and report
// the status to provided DiscoveryListener object.
controller.searchForDevices(new DiscoveryListener() {
      @Override
      publicvoid onDiscoveryStarted() {
            // controller started searching for devices
      }

      @Override
      publicvoid onDiscoveryFinished() {
            // controller finished device search or it was interrupted
      }

      @Override
      publicvoid onDeviceFound(CNPDevice<?, ?> device) {
            // valid device has been found
      }
});
```

```
// Interrupts current device search if there is any being performed
controller.stopSearchingForDevices();

// Returns list of known/bound devices. This method can be called without
// performing search.
controller.getBoundDevices();
```

CNPDevice object obtained this way can then be passed to `CNPController.connectToDevice()` method.

4.5.2 Payment events handling (CNPListener)

Transaction processing is controlled by implementing the CNPListener interface and reacting to events provided to it. Events are grouped into enums based on their origin and logic group. Some of them also contain action codes, giving information about the circumstances in which a given event has been dispatched.

Location: `de.wirecard.accept.sdk.cnp.observer.CNPListener`

The following code snippet provides a description of the interface methods of such an interface:

```
// get instance of current CNPController
CNPController<?>controller = AcceptSDK.getCNPController();

// register listener object, which will receive events from controller
controller.setCNPListener(new CNPListener() {
	publicvoid onProcessUpdate(ProcessState processState) {
		// handle state change in transaction.
		// See ProcessState enum description for more information
	}



	publicvoid onProcessStarted() {
		// Transaction processing started. This event follows calling
		// CNPController.startTransaction() method (as only c'n'p
		// device acknowledges this).
	}

	publicvoid onProcessFinished(ProcessResult processResult) {
		// Transaction processing on c'n'p device has finished with
		// success or failure. See ProcessResult enum description
		// for more information.
	}

	publicvoid onConnectionStarted() {
		// Follows calling CNPController.connectToDevice() method
		// as only controller acknowledges this instruction.
	}

	publicvoid onConnectionEstablished(boolean restartRequired) {
		// Called as only connection is established and both devices
		// are put into standby mode.
		//
```

```
            // If restartRequired parameter is true, then
                  // CNPController.restartDevice() to be called (It is expected
that
            // process update with RESTARTING value will be dispatched).
            // If c'n'p device doesn't support software restart, it needs
            // to be restarted physically.
            // Ignoring this might lead to unexpected behavior
            // or even security risks.
      }

      publicvoid onAdapterEvent(AdapterEvent adapterEvent) {
            // informs about changes in adapter state. Meaning of specific
                  // values may vary depending on the type of terminal used.
      }
});

// unregistering listener. no more events will be received.
controller.setCNPListener(null);
```

4.5.3 Example CNPListener implementation

This section shows, in greater detail, what actions are expected on different events. It also shows example handling of all values.

```
@Override
publicvoid onAdapterEvent(AdapterEvent event) {
      switch(event) {
      caseADAPTER_DISABLED:
            controller.enableAdapter();
            break;
      caseADAPTER_DISABLING:
            showAdapterDisablingScreen();
            break;
      caseADAPTER_ENABLING:
            showAdapterEnablingScreen();
            break;
      caseADAPTER_IDLE:
            switch(event.getActionCode()) {
            case AdapterEvent.ACTION_IDLE_CONNECTION_FAILED:
                  showConnectingFailedScreen ();
                  break;
            case AdapterEvent.ACTION_IDLE_CONNECTION_FINISHED:
                  showSelectDeviceScreen();
                  break;
            case AdapterEvent.ACTION_IDLE_CONNECTION_LOST:
                  showBluetoothConnectionLostScreen();
                  break;
            case AdapterEvent.ACTION_IDLE:
                  showSelectDeviceScreen();
                  break;
            case AdapterEvent.ACTION_IDLE_CONF_UPDATE_FAILED:
                  showConnectingFailedScreen();
                  break;
            }
            break;
      }
}
```

```
@Override
publicvoid onConnectionStarted() {
	// view indicator informing user about connecting process being performed
	showConnectingScreen();
}

@Override
publicvoid onConnectionEstablished(boolean restartRequired) {
	if(restartRequired == true) {
		AlertDialog.Builder builder = newAlertDialog.Builder(
			BluetoothConnectionActivity.this)
			.setTitle("Restart required")
			.setMessage("Terminal configuration update finished. Terminal
"+
					"will now restart in order to apply
				changes")
			.setPositiveButton("OK", new DialogInterface.OnClickListener()
{
				publicvoid onClick(DialogInterface dialog, int id) {
					controller.restartDevice();
				}
		});
		builder.create().show();
	} else {
		controller.startTransactionProcess(CardEntryType.SWIPE_OR_ICC,
			AcceptSDK.getGratuityType() == GratuityType.TIP);
		}
	}

	@Override
	publicvoid onProcessStarted() {
		// show screen with advice to insert or swipe card.
		showPromptCardEntryScreen();
	}

	@Override
	publicvoid onProcessUpdate(ProcessState state) {
		switch(state) {
		caseCARD_DATA_READ:
			// show screen with advice to follow instructions on terminal
			// display
			showFollowInstructionScreen();
			break;
		caseRESTARTING:
			controller.disconnect();
			break;
		caseSIGNATURE_REQUIRED:
			showSignatureScreen();
			break;
		caseSWIPE_OR_INSERT:
			showSwipeCardScreen();
			break;
		caseTERMINATING:
			showTerminatingScreen();
			break;
		casePREAUTHORIZATION_REQUIRED:
```

```
                    if(state.getActionCode() ==
ProcessState.ACTION_PREAUTHORIZATION_FIRST) {
                          // method description below
                          startPreauthorization();
                    } elseif(state.getActionCode() ==
ProcessState.ACTION_PREAUTHORIZATION_RETRY) {
                          // It is secure to restart preauthorization. Show screen
                          // with button calling startPreauthorization() method to
                          // let attendant decide if preauthorization should
                          // be restarted
                          showRetryPreauthorizationScreen();
                    }
                    break;
             casePROMPT_REMOVE_CARD:
                    // show advice to remove card from terminal
                    showPromptRemoveCard();
                    break;
             }
       }

       @Override
       publicvoid onProcessFinished(ProcessResult result) {
             switch(result) {
             caseSUCCESS:
                    // method description below
                    finalizePayment();
                    break;
             caseVOID:
                    // transaction failed and cannot continue. Might be called
                    // as a result of risk management or validation performed
                    // by terminal
                    showCardDeclinedScreen();
                    break;
             }
       }

       @Override
       publicvoid onTerminalEvent(TerminalEvent action) {
             switch(action) {
             caseCONFIG_UPDATE_FINISHED:
                    // configuration updated successfuly
                    break;
             caseCONFIG_UPDATE_STARTED:
                    // configuration update started
                    break;
             caseFIRMWARE_UPDATE_AVAILABLE:
                    // firmware update is available, but it won't be applied
                    // this time
                    break;
             caseFIRMWARE_UPDATE_FINISHED:
                    // firmware updated successfully
                    break;
             caseFIRMWARE_UPDATE_STARTED:
                    // firmware update started. Don't disconnect from terminal,
                    //as it might cause damage.
                    break;
       }
```

```
}

publicvoid startPreauthorization() {
	showPreauthorizingScreen();
	AcceptSDK.preauthorizePayment(newOnRequestFinishedListener<PreauthResult>()
{
		@Override
		publicvoid onRequestFinished(ApiResult apiResult, PreauthResult
result)
		{
			if(apiResult.isSuccess()) {
			// nothing to do here, since PreauthResult is automatically
			// passed to terminal. Event will be passed to CNPListener
		once
			// result is processed by terminal
			} else {
				Code code = apiResult.getCode();
				if(code == Code.WirecardError) {
					if(apiResult.getWirecardError() != null) {
						WCDCode
wcdCode=apiResult.getWirecardError();
						if(wcdCode != null) {
						// It is important to notify merchant about
						// exact reason of card rejection.
						// currently sdk comes with wcdstrings.xml
						// file containing all possible error
					codes.
							showMessage(
								R.string.wirecard_error,
							wcdCode.getDescriptionResID());
						}
					}
					(...)
				}
				(...)
			}
		}
	});
}

publicvoidfinalizePayment() {
	showSendingPaymentScreen();
	AcceptSDK.sendPayment(newOnRequestFinishedListener<Payment>() {
		@Override
		publicvoid onRequestFinished(ApiResult apiResult, Payment result)
		{
		// standard sendPayment method handling.
		(...)
		}
	});
}
```

4.6 Events handling and communication with Swiper

There is interfaces that is required to properly implement the swiper:
- OnSwipeDetectedListener, which should be used for controlling the flow of the transaction and to react to specific swiper events.

Their detailed description can be found in subsections 4.6.1.
Different implementations of OnSwipeDetectedListener should not require implementation of more listeners than those presented here. It is safe to assume that implementing them is enough to handle payments with different swipers.

4.6.1 Example of OnSwipeDetectedListener implementation (Swiper)

Transaction processing is controlled by implementing the OnSwipeDetectedListener interface and reacting to events provided to it. Events are grouped into enums based on their origin and logic group. Some of them also contain action codes, giving information about the circumstances in which a given event has been dispatched.

Location: `de.wirecard.accept.sdk.swiper.OnSwipeDetectedListener`

The following code snippet provides a description of the interface methods of such an interface:

```
AcceptSDK.registerOnSwipeDetectedListener(this);
AcceptSDK.initSwiper();

@Override
public void onSwipeSuccessful() {
    //sending data to the
        startPreauthorization()
 }

@Override
public void onSwiperEvent(final SwiperEvent state, final Object tag, final
String details) {
    switch (state) {
        case READY:
            makeReady(R.string.wl_general_please_insert_or_swipe_the_card);
            break;

        case READY_FOR_SWIPE:
            makeReady(R.string.wl_general_please_swipe_the_card);
            break;

        case READY_FOR_INSERT:
            makeReady(R.string.wl_general_please_insert_the_card);
            break;

        case CHARGING_UP:
            makeInitializing(R.string.acceptsdk_swiperEvent_charging_up,
tag);
            break;

        case FAILED_SWIPE:
            makeFailedSwipe();
            break;

        case AUTO_CONFIG:
            makeInitializing(R.string.acceptsdk_swiperEvent_auto_config,
tag);
            break;

        case DISCONNECTED:
```

```
            makeDisconnected();
            break;

        case CONNECTED:
            makeInitializing(R.string.acceptsdk_swiperEvent_connected,
null);
            break;

        case UNRECOGNIZED_SWIPER:
            makeDisconnected();
            break;

        case REQUEST_ONLINE_PROCESSING:
            startPreauthorization()
            break;

        case SELECT_AID:
            showSelectAIDDialog((java.util.List<String>) tag);
            break;

        case FINAL_CONFIRM:
            //showFinalConfirmPaymentDialog();
            AcceptSDK.finalConfirm(true);
            break;

        case EMV_STARTED:
            makeEmvFlowStarted();//display processing
            break;

        case TRANSACTION_RESULT:
            TransactionResult transactionResult = (TransactionResult) tag;
            if (transactionResult == TransactionResult.APPROVED) {
                if (successfulPaymentObject != null) {// we have payment
finished, need just show success and send TC.
                    //finalize payment
                } else {
                    Log.e(TAG, "onSwipeEvent with TRANSACTION_RESULT
successfulPaymentObject is null");
                    startReversal(false,null);
                }
            } else {

showError(EMVSwiperUserFeedbackHelper.getStringResForTransactionResult(tran
sactionResult));
                //if terminal and card decided to decline approved
                if (transactionResult == TransactionResult.DECLINED) {
                    startReversal(false,
getResources().getString(EMVSwiperUserFeedbackHelper.getStringResForTransac
tionResult(transactionResult)));
                }

                if (transactionResult ==
TransactionResult.ICC_CARD_REMOVED) {
                    new Handler().postDelayed(
                            new Runnable() {
                                @Override
                                public void run() {

statusTextView.setText(EMVSwiperUserFeedbackHelper.getStringResForTransacti
```

```
onResult(TransactionResult.DECLINED));
                                    startReversal(false,
getResources().getString(EMVSwiperUserFeedbackHelper.getStringResForTransac
tionResult(TransactionResult.DECLINED)));
                                }
                            }, 1000);//for check result message and then
finish
                }
            }
            break;

        case ERROR:
            final EMVError emvError = (EMVError) tag;
            backWithError = true;

showError(EMVSwiperUserFeedbackHelper.getStringResForEMVError(this,
emvError, details));
            break;

        case DEVICE_INFO:

statusTextView.setText(R.string.acceptsdk_swiperEvent_connected);
            //DeviceInfo deviceInfo = (DeviceInfo)tag;
            break;

        case MESSAGE:
            if (tag == null) {
                statusTextView.setText(details);
                break;
            }

            if (tag instanceof DisplayText) {

if(DisplayText.TRANSACTION_TERMINATED.name().equalsIgnoreCase(((DisplayText
) tag).name())){
                    if
(!TextUtils.isEmpty(AcceptSDK.getLastTransactionID())) {
                        startReversal(true, null);
                    } else {
                        makeDisconnected();
                    }
                    return;
                }

                int code =
EMVSwiperUserFeedbackHelper.getStringResForDisplayMessage((DisplayText)
tag);
                if (code > 0) //can be -1 if will not display
                    statusTextView.setText(code);
            } else if (tag instanceof EMVError) {

statusTextView.setText(EMVSwiperUserFeedbackHelper.getStringResForEMVError(
this, (EMVError) tag, details));
            } else if (tag instanceof TransactionResult) {

statusTextView.setText(EMVSwiperUserFeedbackHelper.getStringResForTransacti
onResult((TransactionResult)tag));
            }
            break;
```

```
        default:
            Log.e(TAG, "onSwiperEvent unhandled state occurred");
    }
}

publicvoid startPreauthorization() {
      showPreauthorizingScreen();
      AcceptSDK.preauthorizePayment(newOnRequestFinishedListener<PreauthResult>()
{
            @Override
            publicvoid onRequestFinished(ApiResult apiResult, PreauthResult
result)
            {
                  if(apiResult.isSuccess()) {
                  // nothing to do here, since PreauthResult is automatically
                  // passed to terminal. Event will be passed to CNPListener
            once
                  // result is processed by terminal
                  } else {
                        Code code = apiResult.getCode();
                        if(code == Code.WirecardError) {
                              if(apiResult.getWirecardError() != null) {
                                    WCDCode
wcdCode=apiResult.getWirecardError();
                                     if(wcdCode != null) {
                                     // It is important to notify merchant about
                                     // exact reason of card rejection.
                                     // currently sdk comes with wcdstrings.xml
                                     // file containing all possible error
                                    codes.
                                          showMessage(
                                                R.string.wirecard_error,
                                               wcdCode.getDescriptionResID());
                                    }
                              }
                              (...)
                        }
                        (...)
                  }
            }
      });
}

public void finalizePayment() {
      showSendingPaymentScreen();
      AcceptSDK.sendPayment(newOnRequestFinishedListener<Payment>() {
            @Override
            publicvoid onRequestFinished(ApiResult apiResult, Payment result)
            {
            // standard sendPayment method handling.
            (...)
            }
      });
}
```

4.7 Processing payment with Thyron(Chip&Pin)

This section should be treated as a guideline on how to perform transactions using CNPController. It shows the standard flow of such a transaction using magstripe or ICC interface. The order in which subsections are presented reflects the expected flow during the transaction.

4.7.1 Accessing CNPController instance.

In order to obtain a CNPController object, call `AcceptSDK.getCNPController()` method. If the returned reference is null, make sure that all steps shown in section 1 of this document were performed.

After obtaining a CNPController object, a listener should be registered to start receiving events. Upon registering the listener, the most recent event is being delivered instantly. Thanks to this, it is possible to unregister the listener in onPause method of one activity and register the listener in a different activity without losing track of what has happened in this gap.

4.7.2 Connection

Connecting to device can be performed only by calling `CNPController.connectToDevice()` method, giving an instance of CNPDevice class as a parameter. The CNPDevice can be obtained in 3 ways:
from list returned by `CNPController.getBoundDevices()` method,
by performing search using CNPController.searchForDevices() method and passing DiscoveryListener object to obtain found devices,
by calling `CNPController.getLastConnectedDevice()` method, which will return null if no connection was established yet, or an instance of most recent CNPDevice with which the smartphone has successfully connected.

A connection attempt should be performed only when the Adapter is in *`IDLE`* state.

To enable the adapter, call `CNPController.enableAdapter()` method. In order to disable it, call `CNPController.disableAdapter()` method.

After calling `CNPController.connectToDevice()` method, the following behavior is expected:
`CNPListener.onConnectionStarted()` method is called.
If connection fails, `CNPListener.onAdapterEvent()` method is called with *`IDLE`* value and action code *`ACTION_IDLE_CONNECTION_FAILED`*
If connection fails because update of configuration or firmware was unsuccessful, `CNPListener.onAdapterEvent()` method is called with *`IDLE`* value and action code *`ACTION_IDLE_CONF_UPDATE_FAILED`*
Connection succeeds and `CNPListener.onConnectionEstablished()` method is called with its boolean parameter having 'true' value, meaning that a restart is required. `CNPController.restartDevice()` should be called if the controller supports software restart. If it does not, the c'n'p device should be restarted physically (for example by using a hardware switch).
Connection succeeds and `CNPListener.onConnectionEstablished()` method is called with its boolean parameter having 'false' value, meaning that a restart is not required. The controller is now in standby mode and waiting for the transaction to start.

4.7.3 Configuration update

Chip and PIN devices require maintenance in order to keep their configuration up to date. When possible, AcceptSDK automates this process. In that case a c'n'p device is reconfigured right after establishing a connection on communication layer and before starting a transaction. Keep in mind that this may vary between different controller implementations. In general, the configuration version should be checked before `CNPListener.onConnectionEstablished()` event is dispatched, and updated

accordingly. If this event's boolean parameter has 'true' value, it most likely means that the configuration has been updated and a restart is required to make sure that it is reloaded.

4.7.4 Transaction initialization

After establishing the connection, transaction processing can finally be started. The following example shows how to start a transaction immediately after connecting to the Chip'n'PIN device:

```
@Override
publicvoid onConnectionEstablished(boolean restartRequired) {
	if(restartRequired == true) {
		AlertDialog.Builder builder = newAlertDialog.Builder(
			BluetoothConnectionActivity.this)
			.setTitle("Restart required")
			.setMessage("Terminal configuration update finished. Terminal
"+
					"will now restart in order to apply
				changes")
			.setPositiveButton("OK", new DialogInterface.OnClickListener()
{
				publicvoid onClick(DialogInterface dialog, int id) {
					controller.restartDevice();
				}
		});
		builder.create().show();
	} else{
		controller.startTransactionProcess(CardEntryType.SWIPE_OR_ICC,
				AcceptSDK.getGratuityType() == GratuityType.TIP);
	}
}
```

As you can see, this example also shows how to handle a case when the controller signals the need for device restart.
In case a restart is not required, the transaction process is started with the following setup:
device should ask for a card swipe or icc card entry based on its own rules set,
if device is capable, it should allow entry of tip if it is set as current gratuity type in SDK. It is left to programmer to decide if the Chip'n'PIN device should prompt for a tip or if a tip will be entered in the mobile app itself when the AcceptSDK's gratuity type is set accordingly.

After starting the transaction It is expected to receive onProcessUpdate event with *SWIPE_OR_INSERT* value.

4.7.5 Preauthorization

At some point in transaction processing, the controller will dispatch onProcessUpdate event with *PREAUTHORIZATION_REQUIRED* value, indicating that a preauthorization request needs to be sent. The preauthorization request is not sent automatically, in order to give more flexibility for the programmer implementing SDK.

It is safe to retry preauthorization attempts in case a network error has occurred. If preauthorization failed for another reason, it is advised to show to the merchant why the request has failed. This is because the returned error may include such information as: this specific card should not be honored; it is stolen; or the transaction amount exceeds the amount available on that card.

This is possible using the following calls:

```
AcceptSDK.preauthorizePayment(new OnRequestFinishedListener<PreauthResult>() {
```

```
	@Override
	publicvoid onRequestFinished(ApiResult apiResult, PreauthResult result) {
		(...)
		if(apiResult.getWirecardError() != null) {
			WCDCode wcdCode = apiResult.getWirecardError();
			if(wcdCode != null)
				showMessage(wcdCode.getDescriptionResID());
		}
	}
}
```

Where showMessage is the propertiary method taking string resource id as parameter. Messages contained in strings resources connected with those error codes are currently available only in English language.

4.7.6 Signature placing

After passing PreauthResult object to controller via `CNPController.setPreauthorizationResult()` method, the controller may take the decision of requesting a signature to verify the cardholder. At this point, `CNPListener.onProcessUpdate()` method will be called with *`SIGNATURE_REQUIRED`* value. Depending on the type of CNP device used, it may be required to state if the signature is valid using its interface (for example, confirm that the signature is valid by pressing a button). Because of this, a signature should be provided to the SDK only after `CNPListener.onProcessFinished()` method is called with *`SUCCESS`* result and before attempting to capture the transaction.

There are currently 2 ways to provide a signature image to the SDK:
call `AcceptSDK.setPaymentSignature()` method, giving path to signature image present in local storage. This image will be then attached to the capture request.
call `AcceptSDK.setSignatureBytes()` method, providing byte array containing binary data of jpeg or png compressed image. This way is more secure than first one, since signature doesn't have to be stored as image in local storage.

4.7.7 Transaction finalization (capture)

The final step of performing the transaction using CNPController is capturing it. The amount specified in transaction will be subtracted from client's account only after capture. Capture should be done after receiving `CNPListener.onProcessUpdate()` event with *`CAPTURE_REQUIRED`* value. Performing capture on a transaction that was not preauthorized, or that failed preauthorization (on backend side or was rejected by c'n'p device), will automatically fail.

To send a capture request, call `AcceptSDK.sendPayment()` according to the description contained in section 3.6.

If sendPayment request succeeds, it can be safely assumed that the transaction was completed without errors.

4.7.8 Interrupting transaction

In case a transaction needs to be terminated, there are a number of available options:
canceling transaction on Chip'n'PIN device. However, this is not always possible.
calling `CNPController.voidTransaction()` method. This starts cancelling the procedure, which based on device type, might take a few seconds.
calling `CNPController.disconnect()` method. When `disconnect()` is called, the communication link is still maintained for a short period of time in order to void the transaction gracefully.

4.8 Connection and transaction flow diagrams

The goal of this section is to provide better insight into the processing done between the terminal and the application. The diagrams contain information about possible flows and show what conditions must be met to receive specific events.

The following scheme is used in both diagrams:

VALUE onAdapterEvent() is dispatched with *AdapterEvent.VALUE*

VALUE onProcessUpdate() is dispatched with *ProcessState.VALUE*

VALUE onProcessFinished() is dispatched with *ProcessResult.VALUE*

methodName other event is dispatched by provided methodName.

ACTION_VALUE – indicates that an event is dispatched with a specified action

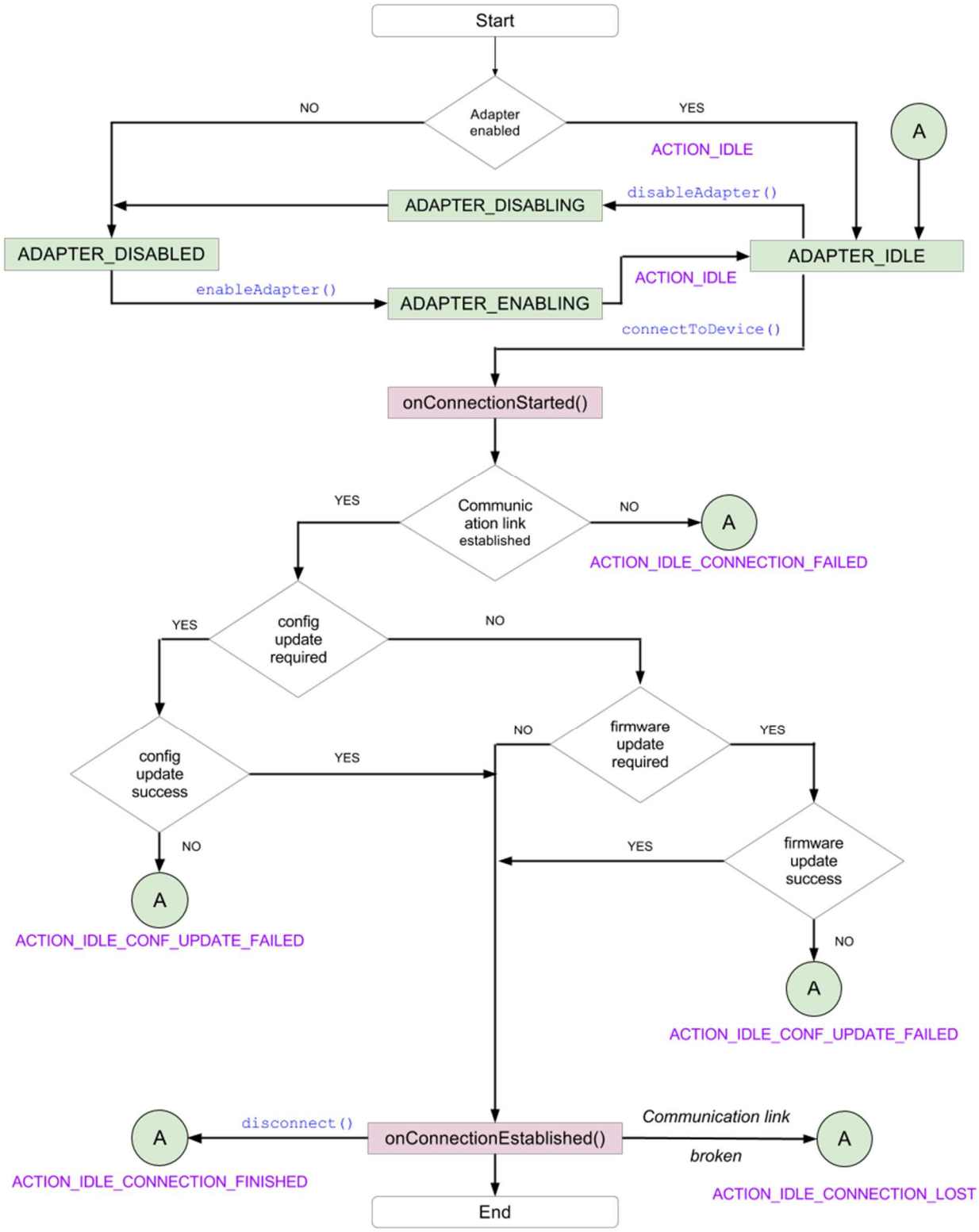
methodName() – indicates that transition to another state is initiated by calling *CNPController.methodName()* method.

methodName() – indicates that transition to another state is initiated by calling *AcceptSDK.methodName()* method.

External event – indicates that additional actions are required to change the state, such as interaction with GUI or terminal.

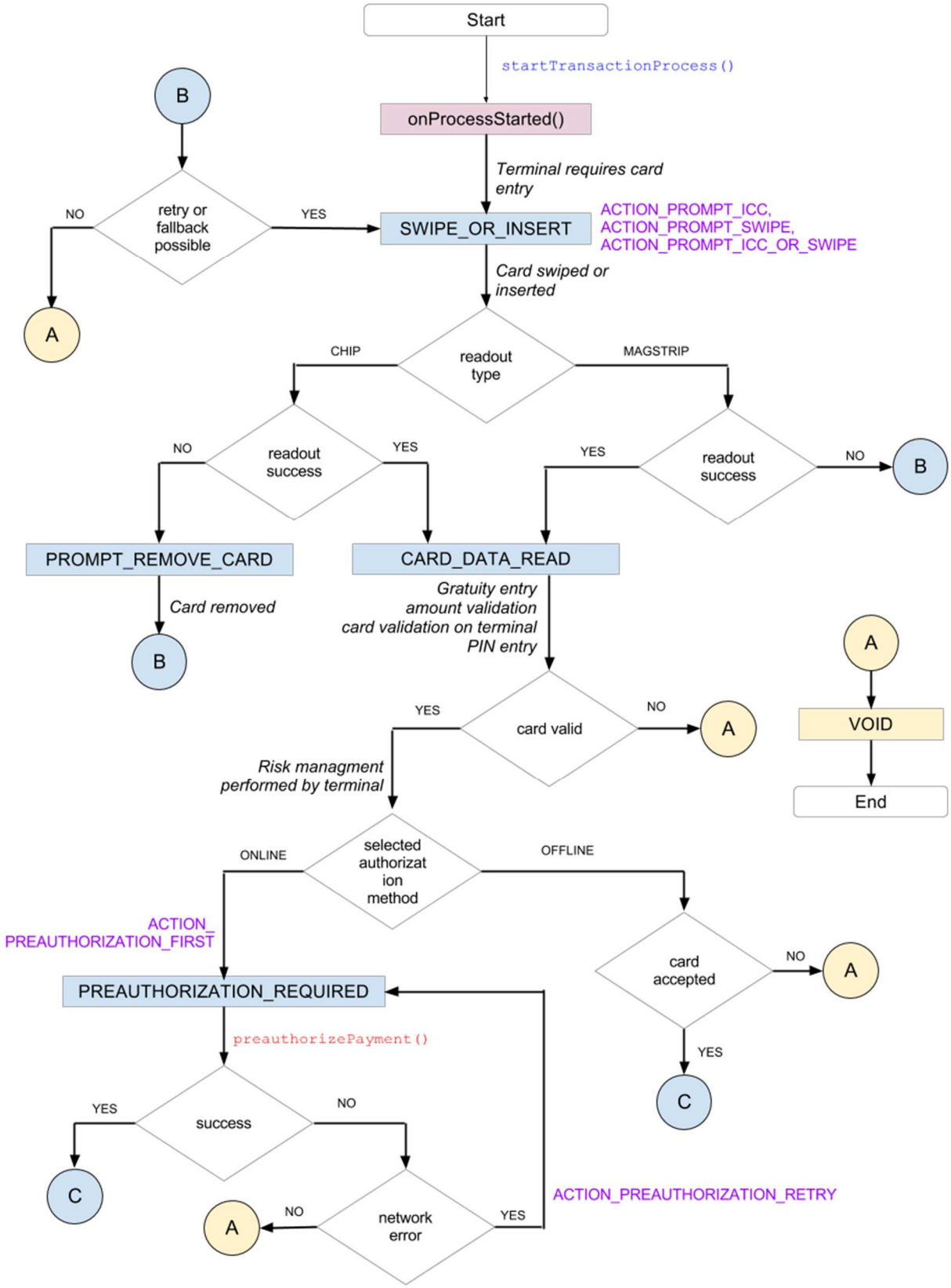
The most recent event is preserved in CNPController. As a result, after unregistering CNPListener, the latest event is dispatched to it as soon as it is registered again (this does not have to be the last event received before unregistering, as a new event could arrive before registering again). Because of this persistent nature, events are shown on diagrams as states.

For internal use only



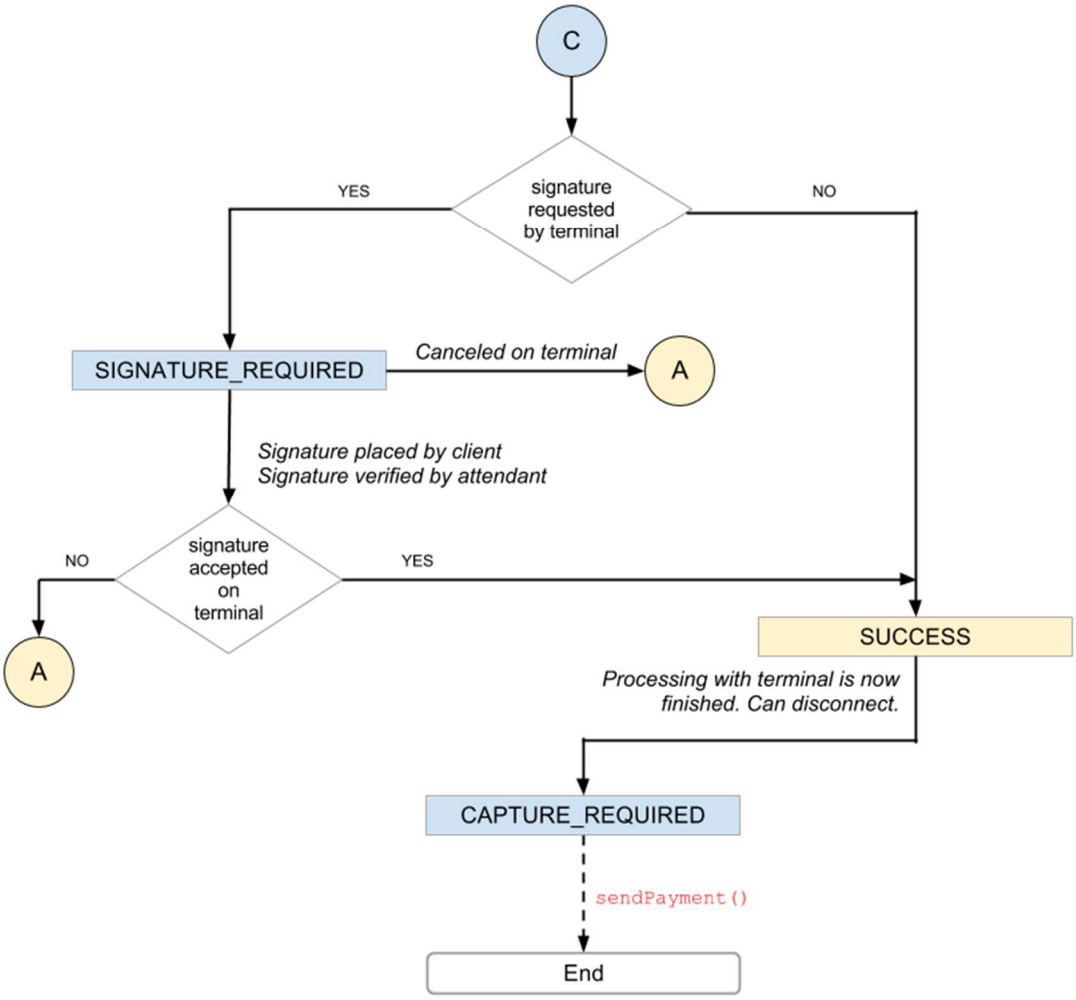
Connection flow diagram

For internal use only



Transaction flow diagram. Continuation on next page.

For internal use only



Transaction flow diagram cont.

4.9 Processing payment with Swiper (chip&sign)

This section should be treated as a guideline on how to perform transactions using Swiper OnSwipeDetectedListener. It shows the standard flow of such a transaction using magstripe or ICC interface. The order in which subsections are presented reflects the expected flow during the transaction.

4.9.1 Accessing Swiper instance.

In order to obtain a Swiper object, call `AcceptSDK.getSwiper()` method. If the returned reference is null, make sure that all steps shown in section 1 of this document were performed.

After obtaining a Swiper object, a OnSwipeDetectedListener listener should be registered to start receiving events. Upon registering the listener, the most recent event is being delivered instantly. Thanks to this, it is possible to unregister the listener in onPause method of one activity and register the listener in a different activity without losing track of what has happened in this gap. Communication to Swiper terminal is provided by the audio jack. It is good to call initSwiper directly in the onCreate of activity.

```
AcceptSDK.registerOnSwipeDetectedListener(this);
```

4.9.2 Tip and gratuity

Swiper device is not capable to display some messages, it is using *`onSwiperEvent`* (Message) to ingorm about statuses. Allowing entry of tip if it is set as current gratuity type in SDK. It is left to programmer to decide if the applicatin should prompt for a tip. Tip will be entered in the mobile app itself when the AcceptSDK's gratuity type is set accordingly. This setting should be done in time of initialization of communication with BBPos Swiper described in 4.9.3.

4.9.3 Connection

Connecting to device should be performed in two steps:

1. Physicaly connect Swiper terminal to audiojack
2. by calling `AcceptSDK.initSwiper()` method, which is initialisating communication with terminal.

PlugInAction is method with couple of commands needed to start communication and receive fist response *`onSwiperEvent(SwiperEvent.DEVICE_INFO,TerminalInfo,message).`* Last command is starting “CheckCardMode” with request for displaying corresponding message.

If in the step 2. of connection process is terminal not present, plugin action will be fired by the pluging terminal to the smartfone. Wecan write that registering of listener and call *`initSwiper`* is everything what is required to implement proper initialization in android activity.

In this time Sdk is controlling batteryLevel state, and if it is lower like 10%, communication will stop with *`EMVError.LOW_BATTERY`* message.

4.9.4 Autoconfiguration

Swiper devices require maintenance in order to keep their configuration compatible with android device used for communication with backend. AcceptSDK and terminal SDK automates this process with starting autoconfiguration process. This is according to hardware compatibility and setup some maximal and minimal configuration values, because terminal is communicating with smartpfone per stereo jack. AcceptSDK provides functionality to switch on automatic configuration process.

Is using local parameter defined in SDK : *R.bool.wl_auto_configuration_enabled*. After autoconfiguration AcceptSDK remember and save configuration profile for next use. Every next try to connect terminal and start transaction is this configuration profile used. This is saving time, because autoconfiguration is quit long process.
On end of autoconfiguration procces we have device asking for a card swipe or icc card entry based on its own rules set.

4.9.5 MRC vs.start EMV

Last state was perompting for swipe or insert card.
Depends on card type with specified payment method (pure magstripe or chip and magstripe or pure chip) can terminal response with *onSwipeSuccessful()*or can start EMVprocess.
It is depending of payment method used. If user decide for magstripe AcceptSDK will provide *onSwipeSuccessful().* If used chip card AcceptSDK started EMV transaction.

4.9.6 Configuration update

In case of EMV transaction AcceptSDK starts automatically download terminal configuration from backend and start EMV transaction with this configuration like with one of parameter needed to provide correct EMV data for sendig to the backend.(4.9.7)

4.9.7 Preauthorization

At some point in transaction processing, the controller will dispatch using dispatchSwiperEvent() event with *SwiperEvent.REQUEST_ONLINE_PROCESSING* value, indicating that a preauthorization request needs to be sent. The preauthorization request is not sent automatically, in order to give more flexibility for the programmer implementing SDK.

It is safe to retry preauthorization attempts in case a network error has occurred. If preauthorization failed for another reason, it is advised to show to the merchant why the request has failed. This is because the returned error may include such information as: this specific card should not be honored; it is stolen; or the transaction amount exceeds the amount available on that card.

This is possible using the following calls:

```
AcceptSDK.preauthorizePayment(new OnRequestFinishedListener<PreauthResult>() {
	@Override
	publicvoid onRequestFinished(ApiResult apiResult, PreauthResult result) {
		(...)
		if(apiResult.getWirecardError() != null) {
			WCDCode wcdCode = apiResult.getWirecardError();
			if(wcdCode != null)
				showMessage(wcdCode.getDescriptionResID());
		}
		(...)
		AcceptSDK.sendOnlineprocessResult(result.getEmvData())
	}
}
```

Where showMessage is the propertiary method taking string resource id as parameter. Messages contained in strings resources connected with those error codes are currently available only in English language.
Important is to call `AcceptSDK.sendOnlineprocessResult(result.getEmvData())` to provide feedback to terminal and card if it is not empty.

4.9.8 Signature placing

Important information is that swiper can use only one Cardholder Verification Method and it is Signature. Good strategy is to provide signature using *AcceptSDK.setSignatureBytes(byte[] byteArray)* to SDK before goOnlineRequest (if EMV) or after succesfull swipe.
Call *AcceptSDK.setSignatureBytes()* method, providing byte array containing binary data of jpeg or png compressed image. This way is more secure than first one, since signature doesn't have to be stored as image in local storage.
Usually it is after *onSwipeSuccessful()* or *dispatchSwiperEvent()* event with *SwiperEvent.REQUEST_ONLINE_PROCESSING* value. After passing PreauthResult object to swiper via AcceptSDK.*sendOnlineprocessResult(result.getEmvData())* method, the terminal may take the decision of requesting a signature to verify the cardholder (In case of swiper terminal it is everytime).
At this point, we have to show signature in app and provide functionality for aprooving / declining signature comparision check.

4.9.9 Transaction finalization (capture)

The final step of performing the transaction using Swiper is capturing it. The amount specified in transaction will be subtracted from client's account only after capture. Capture should be done after receiving *dispatchSwiperEvent()* event with *SwiperEvent.TRANSACTION_RESULT* with tag *TransactionResult.APPROVED* value.

```
case TRANSACTION_RESULT:
    TransactionResult transactionResult = (TransactionResult) tag;
    if (transactionResult == TransactionResult.APPROVED) {
        //send capture request
        }
```

Performing capture on a transaction that was not preauthorized, or that failed preauthorization (on backend side or was rejected by terminal device), will automatically fail.

To send a capture request, call AcceptSDK.*certifyTransaction(* AcceptSDK.*getLastTransactionID(), AcceptSDK.getLastApplicationCryptogram());*

If sendPayment request succeeds, it can be safely assumed that the transaction was completed without errors.

```
new Thread(new Runnable() {
    @Override
    public void run() {
        final AcceptBackendService.Response<AcceptTransaction, Void>
response =

AcceptSDK.certifyTransaction(AcceptSDK.getLastTransactionID(),
AcceptSDK.getLastApplicationCryptogram());
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                if ( response.hasError() || isApproved(response.getBody()))
{
//show error
                } else {
                    //show success summary
                }
            }
        });
    }
}).start();
```

5 Payment flow

5.1 Payment states

Once a payment is transferred from the SDK to the Accept system, the payment will have one of following states:

- pending
- rejected
- approved
- refunded
- reversed

5.1.1 Pending payment

While payment is in a **pending** state, it has been added to Accept Backend but has not yet completed the transaction with the WD gateway. Since a purchase request is performed immediately, this state is transient. In the event of a failure in the purchase request or the server itself, pending payments will still be synchronized using **QUERY**

The pending state **does not mean** that payment has a PENDING state in WD. A pending payment may not even have representation in WD gateway. It means only that the mobile application has begun the process of performing the payment.

5.1.2 Approved payment

Payments with the **approved** status have been processed, and the WD gateway has returned the state PENDING/ACK. Approved payments can be reversed or refunded, which will subsequently change their state to **reversed** or **refunded** as appropriate. These two states are final states.

5.1.3 Rejected

Payment has been rejected for one of two reasons:

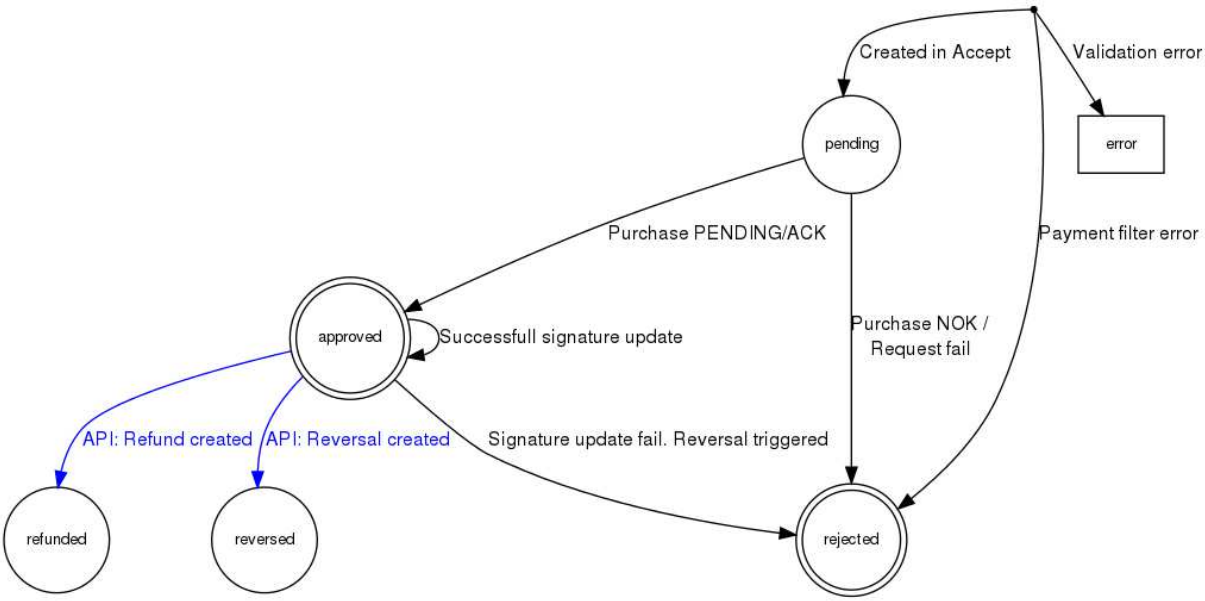
- Rejected instantly by Accept - this means that it will not have representation in WD.
- Rejected by WD Gateway - this means that WD returned the status NOK on purchase or transaction update request.

5.1.4 Refunded

Payment has been fully processed and the related Refund transaction exists in WD in a PENDING/ACK state.

5.1.5 Reversed

Payment has been fully processed and the related Refund transaction exists in WD in a PENDING/ACK state.



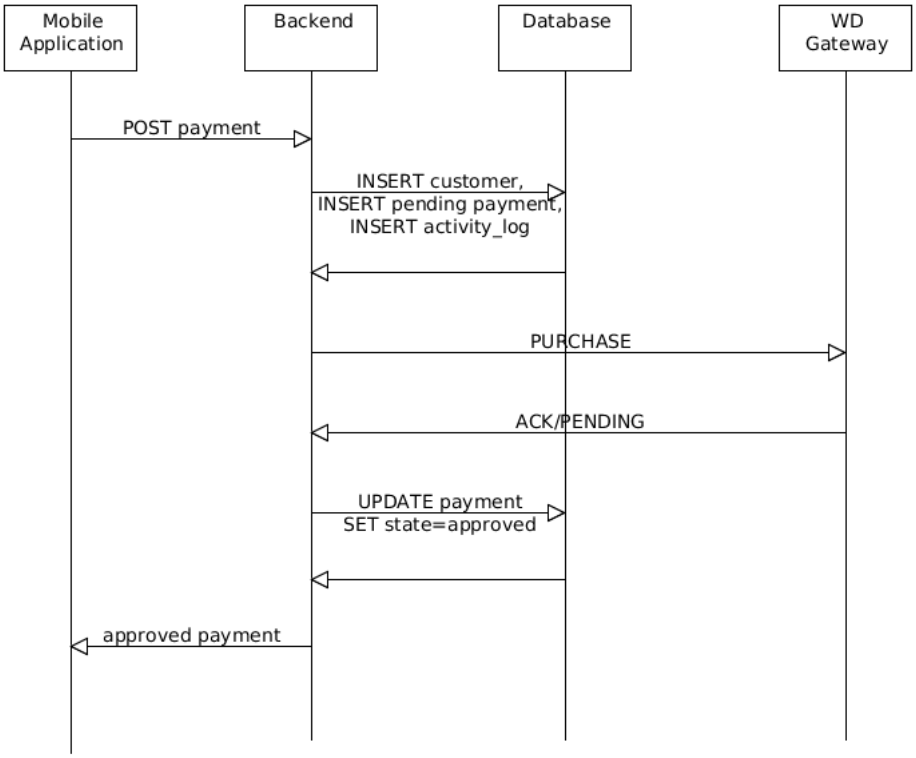
Payment states within Accept backend

5.2 Create payment request

All flow requests to the WD gateway are performed by the Accept Backend. The mobile application makes a POST request with all the necessary data, containing:

- transaction amount
- currency
- encrypted credit card data
- POS data
- PIN data

With this set of data, the backend can perform a request to the WD gateway. A complete successful payment flow is presented in the following diagram:



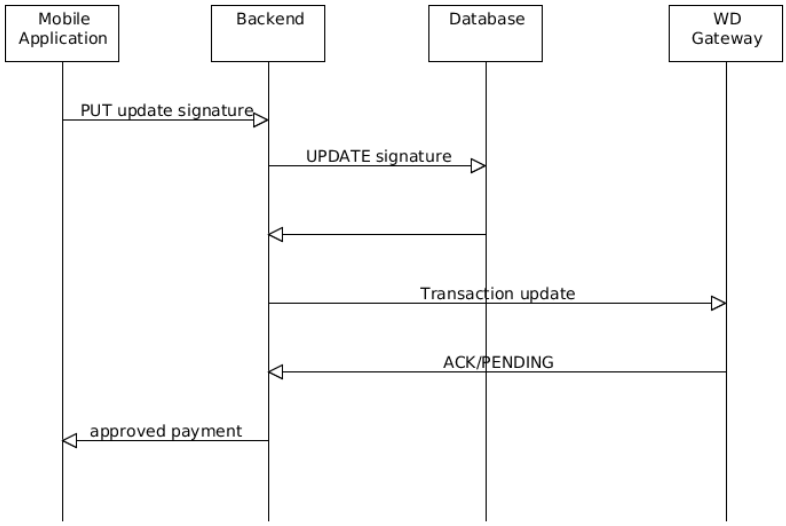
Flow of approved payment

It should be noted that payment is inserted into the database with a 'pending' state before the WD request is made. If an error occurs in a later stage of the flow, this payment will be synchronized with WD using QUERY. This should be treated as an edge case.
When WD returns a PENDING/ACK response, the payment state is updated to 'approved' and returned to the mobile application.

At this point, the Thyron device used for the transaction may decide that a signature is necessary to finalize the payment flow.

5.3 Signature update

If a signature is required, the mobile application sends the signature in .png format to the Accept Backend. The Backend updates the signature in the database and performs a transaction update with the WD gateway. If the response is ACK/PENDING then no additional status update is required, and the payment is returned to the mobile application with an 'approved' state:

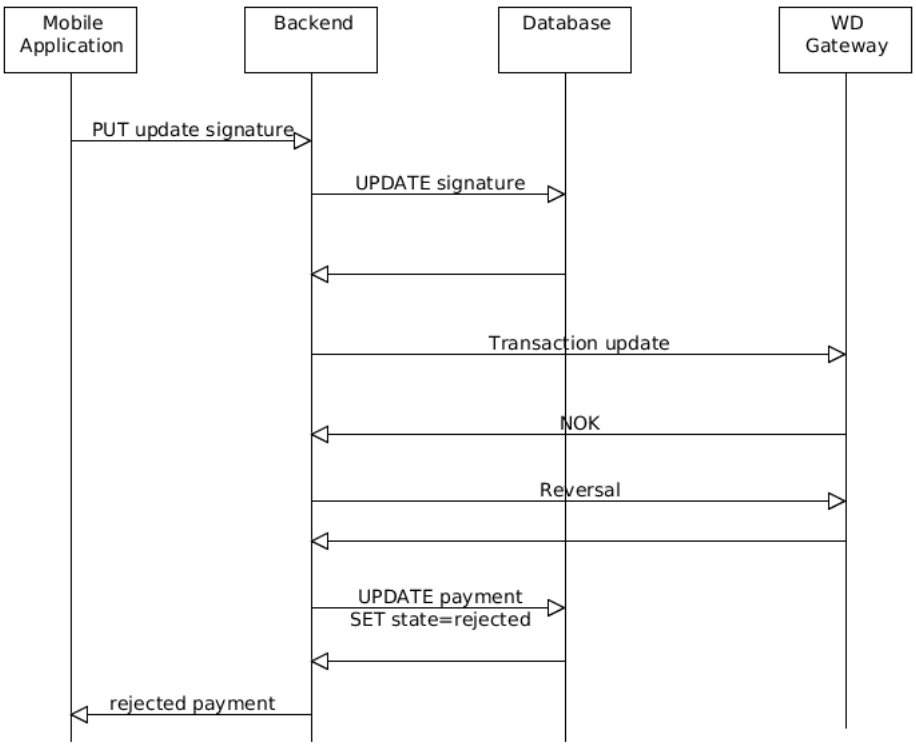


Flow of approved payment with signature update

startTransactionProcess

In the event of a failure during signature update, two additional operations are performed:

- Reversal request is sent to WD gateway
- Payment state is updated to 'rejected'



Flow of rejected payment with signature update rejected