Quick Start Guide

Android version

Author: Wirecard Technologies GmbH

© WIRECARDAG2016
www.wirecard.com | info@wirecard.com

Contents

1 Introduction

This document describes how Accept SDK should be used by Android applications.

1.1 Audience

This document is intended for the developers who integrate Accept SDK into their Android applications.

1.2 Related Documents

1. Accept SDK Android Documentation - Full manual for Android SDK.
2. Public demo app source code repository: https://github.com/WirecardMobileServices/acceptSDK-Android/tree/master/demo/demo renamed from (https://github.com/mposSVK/acceptSDK-Android/tree/master/demo/demo)
3. Public SDK repository: https://github.com/WirecardMobileServices/acceptSDK-Android/tree/master/demo/demo renamed from (https://github.com/mposSVK/acceptSDK-Android)

In this document used as (RD1, RD2, RD3)

1.3 Revision History

SDK Version	Date	Name	Comments
1.6.0	06.12.2017	Marek H.	Added native SDK support for asynchronies FW update, Rupay, CashBack
1.5.10	23.8.2017	Marek H.	Receipt builder, rename bbPOSExtension /BBPosExtension, ThyronExtension /SpireExtension, add paymentAdditionalAtributes
1.5.9	16.7.2017	Marek H.	Alipay, Sepa,Cash payment
1.5.7	06.12.2016	Marek H.	Added relations to GitHub, and GitHub demo source code files
1.4.9	08.01.2016	Patrik M.	Added 6. Step in the section 2: added loading of AID configuration after success initialization / login
1.0.0	06.08.2015	Damian K.	Fixed typos
1.0.0	31.07.2015	Damian K.	Initial version.

Copyright

Copyright © 2008-2015 WIRECARD AG
All rights reserved.

Printed in Germany / European Union
Last updated: June 2015

Trademarks

The Wirecard logo is a registered trademark of Wirecard AG. Other trademarks and service marks in this document are the sole property of the Wirecard AG or their respective owners.
The information contained in this document is intended only for the person or entity to which it is addressed and contains confidential and/or privileged material. Any review, retransmission, dissemination or other use of, or taking of any action in reliance upon, this information by persons or entities other than the intended recipient is prohibited. If you received this in error, please contact Wirecard AG and delete the material from any computer.

2 Overview

Accept SDK for Android is an mPOS solution for Android devices that enables electronic payments through a range of terminals connected using Audio Jack or Bluetooth.

Currently the following terminals are supported:

1. Spire PosMate
2. Spire SPm2
3. BBPOS EMVSwiper

Accept SDK for Android is distributed as a set of AAR libraries or simple as linked dependency in gradle file (related documents) to GitHub repository. It consists of two or more aar files depending on which terminals are used for the payments. The set always contains the core library (*accept-sdk-android-acceptsdksource-release_x.x.x.aar*) and at least one aar-file more, as an extension for the core.
The extensions provide support for the terminals and allow the core to delegate the payment procedure to them. For example, if the application is going to use terminals for the payments, the SDK should have following aar files or links in gradle file:

- accept-sdk-android-acceptsdksource-release_x.x.x.aar
  https://github.com/WirecardMobileServices/acceptSDK-Android/blob/master/accept-sdk-android-acceptsdksource-release.aar

- and one (or both) of extensions files:
  i. accept-sdk-android-extension-spire-release_x.x.x.aar
     https://github.com/WirecardMobileServices/accept-android-extension-spire/blob/master/accept-sdk-android-extension-spire-release.aar
  ii. accept-sdk-android-extension-bbpos-release_x.x.x.aar
     https://github.com/WirecardMobileServices/accept-android-extension-bbpos/blob/master/accept-sdk-android-extension-bbpos-release.aar

2.1 Extension compatibility table

Current extensions-compatibility is written down in table. Current state is on GitHub:

- acceptSDK-Android
- accept-android-extension-spire
- accept-android-extension-bbpos

3 SDK Integration

To integrate the Accept SDK please follow the steps below:

Note: integration using demo app source code repository requires basic knowledge about android project structure, and usage of product flavours.

This part of guide will explain implementation of Spire(Thyron) extension.

BBPos extension implementation is almost same.

- GitHub BBPos demo example

3.1 Using aar files

If your implementation will use direct ("in project" libs folder) link to aar files, you have to manually download aar files from GitHub repository every time if released some new version.
Copy Accept SDK AAR's files into your project directory and modify **build.grade** file:

```
apply plugin: 'com.android.application'

repositories {
    jcenter()
    maven { url "https://jitpack.io" }
    flatDir {
     dirs 'libs'
}

buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:2.3.3'
    }
}

android {
    packagingOptions {
        exclude 'META-INF/notice.txt'
        exclude 'META-INF/license.txt'
        exclude 'META-INF/LICENSE'
        exclude 'META-INF/LICENSE.txt'
        exclude 'META-INF/NOTICE'
        exclude 'META-INF/NOTICE.txt'
        exclude 'META-INF/services/com.fasterxml.jackson.core.JsonFactory'
    }
}

dependencies {
    compile (name: 'accept-sdk-android-acceptsdksource-1.6.0', ext: 'aar')
    compile (name: 'accept-sdk-android-thyronextension-1.6.0', ext: 'aar')
}
```

3.2 Using gradle and dependency

Simply copy gradle file from demo app for use it like maven gradle dependency:
https://github.com/WirecardMobileServices/acceptSDK-Android/blob/master/demo/demo/sample/build.gradle

If using maven it is important to specify maven repository link

```
repositories {
    maven { url "https://maven.google.com" }
    maven { url "https://jitpack.io" }
    flatDir {
        dirs 'libs'
    }
}
```

And dependencies:

```
dependencies {
    compile 'com.fasterxml.jackson.core:jackson-databind:2.8.3'
    compile 'com.github.mposSVK:acceptSDK-Android:1.6.0'
    compile 'com.github.mposSVK:accept-android-extension-spire:1.6.0'
    //compile 'com.github.mposSVK:accept-android-extension-bbpos:1.6.2'
}
```

Note: *Extension compatibility is described here*

3.3 Modify AndroidManifest.xml file

to get the following permissions. This step is optional. Gradle builder merges the permissions from aar into application's manifest.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android">
    ...
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.BLUETOOTH" />
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
    ...
</manifest>
```

3.4 Initialize SDK

Add custom **Application** class to your project and add Accept initialisation procedure to Application class. This part can be written also in custom activity implementation. (Explained detail later)

```
public class Application extends android.app.Application {

    @Override
    public void onCreate() {
        super.onCreate();
        // Init loads the rest of configuration from config_for_accept.xml file.
        AcceptSDK.init(this, "{YOUR CLIENT ID}", "{YOUR CLIENT SECRET}",
                "{YOUR BACKEND INSTANCE URL}");
        AcceptSDK.loadExtensions(this, null);
    }
}
```

Note: *Please obtain Client ID/Secret from Accept support team.*
Everything is coded and commented in related documents.

- GitHub demo example

3.5 Create config file

Add Accept's configuration properties to res/values:

Note: We propose to **create config.xml** with this content in case of Spire(Thyron) extension used.

Every extension is independent, and implementer is able to choose which extension will be used by simply change **"extensions_list"** attribute in config file.
It is easy, compile your app with specified dependencies (gradle file) on extension and in config file simply say that you will use this extension. SDK, to know which extension should be used is using reflection (wiki: "ability of a computer program to examine, introspect, and modify its own structure and behaviour at runtime") for this feature we have to use reserved name "SpireExtension". (SDK versions older as 1.5.10 are using old naming "`ThyronExtension`")
All attributes are self-describing.

```
<!--ThyronExtension is able to switch off contact less payments acceptance-->
<item name="acceptsdk_support_contactless" type="bool">true</item>

<item name="wl_default_terminal_type_use" type="string">SpireExtension</item>

<!-- set list of supported extensions -->
<string-array name="extensions_list">
    <item>SpireExtension</item>
</string-array>
```

Note: Again, everything is in case of BBPos implementation on GitHub.

- GitHub BBPos demo example

3.6 Payment flow

Let us start with basic flow explanation with introduce basic abstract class (controller) methods.

```
public abstract class PaymentFlowController {

    /**
     * discover devices and select one for using,  in case of Spire BT it is required to start first connect because we need base info about BT terminal device
     * @param context
     * @param callbackDelegate
     */
    public abstract void discoverDevices(final Context context, final DiscoverDelegate callbackDelegate);

    public abstract void connectAndConfigure(final Context context ,final Device device, final ConfigureListener listener, final boolean firmwareUpdateAllowed);

    /**
     * use for payment with device,
     * @param device Device - device identification
     * @param amount payment amount
     * @param currency payment currency
     * @param delegate delegate is interface PaymentFlowDelegate, responsible for library status feedback and signature capturing , signature confirmation, errors displaying
     */
    public abstract void startPaymentFlow(final Device device, final long amount, final Currency currency, final PaymentFlowDelegate delegate) throws  IllegalStateException;

    /**
     * cancel payment and prepare SDK to next payment
     */
    public abstract void cancelPaymentFlow();
```

And AcceptThyronPaymentFlowController as controller designed for Spire(Thyron) extension implementation.

3.7 Discover delegate, Device

Discover devices is first step in implementation. From integration, point is important to be familiar with basic interface and model used for data interchange. Using discover delegate you will get list of objects representing device identification used in next steps for payment. If more devices connected, you will get callback event:

onDiscoveredDevices(List<Device> devices, final SelectDeviceDelegate selectDeviceDelegate);

which provides list of devices to be able to decide which should be used for communication / payment. There is possibility to set chosen device by calling:

PaymentFlowController.SelectDeviceDelegate onDeviceSelected(device)

This feature is responsible to set current used terminal / device into SDK, to be able to make future payment.

Note: It is required to implement in *onDiscoveredDevices() method a* picker dialog to provide possibility to select terminal (from list of available) used for payment.

- GitHub demo example

```
public enum DiscoveryError {
    NO_BLUETOOTH_MODULE,
    NO_PERMISSION_TO_USE_BLUETOOTH,
    FAILED_TO_ENABLE_BLUETOOTH,
    NO_PERMISSION_TO_USE_MICROPHONE,
    AIRPLANE_MODE_ON,
    NOTHING_INSIDE_JACK,
    NOTHING_INSIDE_USB,
    UNKNOWN
}

public interface SelectDeviceDelegate {
   void onDeviceSelected(final Device device);
}

public interface DiscoverDelegate {
    void onDiscoveryError(DiscoveryError error, String technicalMessage);
}

public interface SpireDiscoverDelegate extends DiscoverDelegate {
    void onDiscoveredDevices(List<Device> devices, final SelectDeviceDelegate selectDeviceDelegate);
}

public interface BBPosDiscoverDelegate extends DiscoverDelegate {
    void onDiscoveredDevices(List<Device> devices);
}
/**
 * discover devices and select one for using
 * in case of Spire BT it is required to start first connect because we need base info about BT terminal device
 * @param context
 * @param callbackDelegate
 */
public abstract void discoverDevices(final Context context, final DiscoverDelegate callbackDelegate);

public class Device {
    public String displayName;
    public String id;
    public String id_vendor;
    public String id_product;
.
.
}
```

3.8 Payment flow delegate

Second step is to make payment. From integration point of view is important to get familiar witch basic interface used for this functionality.

Payment flow delegate is basic object used for communication during payment. It is handling payment flow state updates, errors during payments, signature verification methods callbacks, etc.

```
public interface SignatureRequest {
    void signatureEntered(byte[] signaturePNGBytes);
    void signatureCanceled();
}

public interface SignatureConfirmationRequest {
    void signatureConfirmed();
    void signatureRejected();
}

public enum ConfigProgressState {
    FW_DOWNLOAD_COMPLETED,
    FW_UPDATE_IN_PROGRESS,
    CONFIG_PROGRESS,
    CONFIGURATION_UPDATE_FILE_COUNTER
}

public interface ConfigureListener {
    void onConfigurationStarted();
    void onConfigurationInProgress(ConfigProgressState state, String message);
    void onConfigureSuccess(boolean restarted);
    void onConfigureError(Error error, String errorDetails);
}

public interface FirmwareUpdateListener {
    void onConfigureError(Error error, String errorDetails);
    void onFirmwareUpdateAvailable();
    void onFirmwareUpdateNotNeeded();
}


public enum Update {
    DATA_PROCESSING,
    LOADING,
    RESTARTING,
    ONLINE_DATA_PROCESSING,
    EMV_CONFIGURATION_LOAD,
    FIRMWARE_UPDATE,
    FIRMWARE_UPDATE_AVAILABLE,
    CONFIGURATION_UPDATE,
    CONFIGURATION_UPDATE_AVAILABLE,
    COMMUNICATION_LAYER_ENABLING,
    TRANSACTION_UPDATE,
    WAITING_FOR_INSERT,
    WAITING_FOR_SWIPE,
    WAITING_FOR_INSERT_OR_SWIPE,
    WAITING_FOR_INSERT_SWIPE_OR_TAP,
    WAITING_FOR_CARD_REMOVE,
    WAITING_FOR_AMOUNT_CONFIRMATION,
    WAITING_FOR_SIGNATURE_CONFIRMATION,
    WAITING_FOR_PINT_ENTRY,
    WRONG_SWIPE,
    TERMINATING,
    DONE,
    UNKNOWN
}

public enum Error {
    COMMUNICATION_FAILED,
    PAIRING_LOST,
    DATA_PROCESSING_ERROR,
    ONLINE_PROCESSING_FAILED,
```

```
    ONLINE_PROCESSING_FAILED_CONNECTION_PROBLEM,
    TRANSACTION_UPDATE_FAILED,
    TRANSACTION_UPDATE_FAILED_CONNECTION_PROBLEM,
    EMV_CONFIGURATION_LOAD_FAILED,
    EMV_CONFIGURATION_LOAD_FAILED_CONNECTION_PROBLEM,
    EMV_CONFIGURATION_INVALID,
    CANCELED_ON_TERMINAL,
    UNKNOWN
}

public interface PaymentFlowDelegate {
    void onPaymentFlowUpdate(Update update);
    void onPaymentFlowError(Error error, String technicalDetails);
    void onPaymentSuccessful(Payment payment, String TC);
    void onSignatureRequested(SignatureRequest signatureRequest);
    void onSignatureConfirmationRequested(SignatureConfirmationRequest signatureConfirmationRequest);
}

/**
 * use for payment with device,
 * @param device Device - device identification
 * @param amount payment amount
 * @param currency payment currency
 * @param delegate delegate is interface PaymentFlowDelegate, responsible for library status feedback and signature
capturing , signature confirmation, errors displaying
 */
public abstract void startPaymentFlow(final Device device, final long amount, final Currency currency, final
PaymentFlowDelegate delegate);
```

3.9 Signature requirements callbacks

Signature is one of most used verification methods for verify card holder during payment.

SDK is saving (sending) signature image during payment into backend database. It is used two interfaces for handle this feature.

First one is for capturing signature. User should provide app to customer and request him to sign transaction directly on smartphone.

```
public interface SignatureRequest {
    void signatureEntered(byte[] signaturePNGBytes);
    void signatureCanceled();
}
```

Second interface is used for terminals without display for verify if signature is matching with one on backside of the payment card. e.g.: Spire terminals are solving this feature locally on terminal display, but BBPos requires implementing this callback to solve signature confirmation feature.

```
public interface SignatureConfirmationRequest {
    void signatureConfirmed();
    void signatureRejected();
}
```

Note: For Spire terminals, you can leave this handler empty and just display message about follow instructions on terminal display.

3.10 Terminal configuration

Online payment process is not so easy and it is normal to have some configuration. Our terminals are using configurations related to terminal capabilities, AID configurations, Merchant category code, Merchant identifiers, Terminal identifiers, Country specification, and ea.
Usually is merchant requested to make one test payment every day morning (just to the state when terminal is asking for applying card), to test if connection is working, or if some configurations are not changed. Terminal configuration download and followed upload on terminal device is part of this first transaction. Usually it is followed by restarting terminal.

Mechanism of updating terminal configuration is hidden in the call:

startPaymentFlow(final Device device, final long amount, final Currency currency, final PaymentFlowDelegate delegate);

Update can happen only once per day if first transaction is proceeded. It is not needed to make some additional implementations even when it is possible to run on event (button click).If it is required, sdk supports also separated configuration request for updating terminal configuration. How to do it, is written and detail commented in AbstractSpireMenuActivity.java (on the end it's just connect to terminal, and if SDK has some new config files, they will be uploaded, if not then nothing happened). SDK will not allow you to make transaction with wrong device configuration.

3.11 Firmware update

In demo app is used strategy to update firmware manually (common solution), on button click.
Wirecard Technologies GmbH is providing every time updated and functional terminal, firmware update is very rare feature. Update process contains two requests and connect to terminal.
First request is simple check request. You will get information into SDK, about latest version of firmware for your configuration. Usually it is the same as you are using, therefore this feature is not so important to be implemented on start of integration. It is highly recommended to use *FirmwareVersionCheckAsyncTask.*
If you receive callback action in *onFirmwareUpdateAvailable,* it means that you have to open some activity, and there update your firmware version using:

*AcceptThyronPaymentFlowController.connectAndConfigure(**final** Context context, **final** Device mDevice, **final** ConfigureListener configureListenerDel, **final boolean** firmwareUpdateAllowed)*

Note: because this is using normal payment connection to terminal and same start of communication, in case that its fist action which you have been done in demo app, it starts also device config upload.(What is correct)

Firmware update uploads one big file, it take few minutes. Compared to Configuration update, which is quite quick, it is upload of more small files. Config update usually requires restart of terminal.
For correct implementation we recommend to simply copy implementation from AbstractSpireMenuActivity.java and ConfigurationAndFirmwareUpdateActivity.java in demo app,

Note: It is good to make every day morning one "test" transaction. This transaction must be not completed to successful state. It is good enough to get state "Insert or swipe card" or "Present, insert or swipe card" in case of contact less supported. If you will be in this state of payment, you can cancel payment, and you are sure that no more configuration will happen for today.

Note: The terminal identification based configuration download is a part of merchant identification on backend. It needs to be configured with administrator. If you are getting some error messages / states before payment starts, please contact Accept support team, if it is not delivered together with the bundle.

4 Payment implementation

This part of document will mention about how to implement step after step all features related to payment. After SDK initialization we can start with login.
On the end of this chapter you can find diagram which is graphically representing everything what will be here described.

4.1 Login

The payment procedure starts from a login operation. It initializes the Accept SDK and validates the account used for the payments:

```
final EditText usernameEditText = (EditText)findViewById(R.id.username);
final EditText passwordEditText = (EditText)findViewById(R.id.password);

final String username = usernameEditText.getText().toString();
final String password = passwordEditText.getText().toString();

AcceptSDK.login(username, password, new OnRequestFinishedListener<Object>() {
   @Override
   public void onRequestFinished(ApiResult apiResult, Object result) {
      if ( apiResult.isSuccess() ) {
         // login success.
         return;
      }
      // handle error: apiResult.getDescription());
   }
});
```

- GitHub demo example

After completes a successful login operation, the SDK gets the access token for further networking together with information about logged merchant. Basic merchant information can be obtained using the following methods:

```
final String merchantName = AcceptSDK.getMerchantName();
final String merchantEmail = AcceptSDK.getMerchantEmail();
final String merchantPhoneNumber = AcceptSDK.getMerchantPhoneNumber();
```

4.1.1 Session timeout

After successful login SDK gets the access token. It is representation of logged user. SDK will create session with (configurable) session timeout time. If this session validity time expired, user should login again. This event should be handled in app by implementation of Broadcast receiver.
Create please broadcast receiver able to catch intent `AcceptSDKIntents.SESSION_TERMINATED.`

```
LocalBroadcastManager.getInstance(this).registerReceiver(receiver, new IntentFilter (AcceptSDKIntents.
SESSION_TERMINATED ));

public void sendLogoutIntentAndGoLogin() {
   AcceptSDK.logout();
   Intent intent = new Intent(this, LoginActivity.class);
   intent.putExtra(BaseActivity.LOGOUT, true).addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
   startActivity(intent);

   intent = new Intent(BaseActivity.INTENT);
   intent.putExtra(BaseActivity.INTENT_TYPE, BaseActivity.TYPE_LOGOUT);
   LocalBroadcastManager.getInstance(this).sendBroadcast(intent);
}

private class SessionTerminatedReceiver extends BroadcastReceiver {
```

```
    @Override
    public void onReceive(Context context, Intent intent) {
Log.e("Session Timeout", "sending Log Out");
        sendLogoutIntentAndGoLogin();
    }
}
```

Note: before every longer transaction is recommended to call session refresh

4.1.2 Session refresh

Session refresh is simple request which “says to backend I am alive, still working in app”

```
if (AcceptSDK.isLoggedIn()) {
    //use this if you will stay on same session, just app is working on longer task
    AcceptSDK.sessionRefresh(new OnRequestFinishedListener<HashMap<String, String>>() {
        @Override
        public void onRequestFinished(ApiResult apiResult, HashMap<String, String> result) {
            if (!apiResult.isSuccess()) {
                sendLogoutIntentAndGoLogin();
            }
        }
    });
}
```

- GitHub demo example

When the SDK did a successful login, a new payment can be triggered using **PaymentFlowController**. The class is an abstraction layer for managing the payment procedure. Depending on which terminal is going to be used for the payment, a particular class of the payment flow controller needs to be created. For example, for the Spire / Thyron terminals, the SDK offers **AcceptThyronPaymentFlowController**.

4.2 Devices discovery

The device discovery procedure returns the list of currently available terminals. In case of Bluetooth terminals, it will return the list of paired Bluetooth devices. For Audio Jack devices or USB connected devices, it will return the list of supported terminals.A sample commented code for device discovery is presented below:

```
paymentFlowController.discoverDevices(this, new PaymentFlowController.SpireDiscoverDelegate () {

    @Override
    public void onDiscoveryError(final PaymentFlowController.DiscoveryError error, final String technicalMessage) {
        //showTerminalDiscoveryError
    }

    @Override
    public void onDiscoveredDevices(final List<PaymentFlowController.Device> devices) {
        if (devices.isEmpty()) {
            //showNoDevicesError
        }
        if ( devices.size() == 1 ) {
            currentDevice = devices.get(0);
        }
        //showTerminalChooser
            currentDevice = selected;
    }
});
```

- GitHub demo device discovery example

4.3 Card payment implementation

The payment procedure is driven by the SDK. SDK notifies the user about major events related to the payment using **PaymentFlowDelegate interface**. The same interface is also used to delegate some of the payment steps to the user of SDK, which are related to customer/merchant interaction. An example implementation of the basic payment flow interface is presented below:

```
AcceptSDK.startPayment();// initialization of new payment in SDK

amountCurrency = Currency.getInstance(AcceptSDK.getCurrency());

Float tax;
if(AcceptSDK.getPrefTaxArray().isEmpty())//if not filled out use "0f"
    tax = 0f;
else tax = AcceptSDK.getPrefTaxArray().get(0);// taxes are defined on backend and requested during communication..pls
use only your "supported" values

//here is example how to add one payment item to basket
AcceptSDK.addPaymentItem(new PaymentItem(1, "", currentAmount, tax));
//for demonstration we are using only one item to be able to fully controll amount from simple UI.

// and now we have to get amount in units from basket(with respect to taxes, number of items....)
final long amountUnits =
AcceptSDK.getPaymentTotalAmount().scaleByPowerOfTen(amountCurrency.getDefaultFractionDigits()).longValue();

//and finally start pay( with given device, pay specified units in chosen currency)
paymentFlowController.startPaymentFlow(device, amount, getAmountCurrency(), this);
//for more info look at PaymentFlowActivity.startPaymentFlow(…) method in demo app source code
```

- GitHub demo example

The beginning of the example shows how the Accept SDK must be configured before starting a new payment. The following sequence should always be performed before starting a new transaction:

1. *AccpetSDK.startPayment()* – it clears the data and starts a fresh payment.
2. *AcceptSDK.addPaymentItem()* – it adds a new item to customer's basket. The final amount is calculated from basket's content.
3. *AcceptSDK.getPaymentTotalAmount()* – getter of total sale amount
4. *paymentFlowController.startPaymentFlow()* – it start the payment, establishes a session with the terminal and goes through the payment.

4.4 CashBack payment implementation

Is very similar as normal terminal-used card payment (see 4.3 Card payment implementation), just SDK requires additional setting, and slightly different method usage:

1. *AccpetSDK.startPayment()* – it clears the data and starts a fresh payment.
2. Use:
   AcceptSDK.setCashBackItem(new CashBackItem("CashBack item note", currentAmount));
   instead of adding payment item normally (AcceptSDK.addPaymentItem())
2. To set up correct CashBack mode use please *AcceptSDK.CashBack* enum
3. And call additional support CashBack method in
   ((AcceptThyronPaymentFlowController)paymentFlowController).startCashBackPaymentFlow(Device device, AcceptSDK.CashBack cashBackType, PaymentFlowDelegate delegate);
   instead of call default `payment method :` *startPaymentFlow(…)*

- GitHub demo example

4.5 Sepa payment implementation

Again, it is similar as normal terminal-used card payment (see 4.3 Card payment implementation), just some specific settings are required.

1. *AccpetSDK.startPayment()* – it clears the data and starts a fresh payment.
2. Important is to set second attribute during construction of payment flow controller to true: *AcceptThyronPaymentFlowController(..., boolean sepaPayment, ...)*
   Default value is false, and this method will set all nesesary setting for you.
3. And call additional support Sepa method in : *((AcceptThyronPaymentFlowController)paymentFlowController).startSepaPaymentFlow(final Device device, long amount, final Currency currency, final PaymentFlowDelegate delegate)* instead of call default payment method : *startPaymentFlow(final Device device, long amount, final Currency currency, final PaymentFlowDelegate delegate)*

- GitHub demo example

4.6 Alipay payment implementation

Alipay Payment is little bit different by the implementation as normal payment using terminal. Alipay payment is not using terminal. Alipay method is very specific. Method is using QR code for authentication of merchant and payment is mostly just about collect data for authorize money transfer. We can specify it as just sending information to Accept backend about some income made by Alipay payment method. On backend side you will have then record of successful cash transaction. It can be implemented by few simple calls.

1. *AccpetSDK.startPayment()* – it clears the data and starts a fresh payment.
2. Set SDK payment mode to Cash by simple: *AcceptSDK.setPaymentTransactionType(AcceptSDK.TransactionType.ALIPAY_PAYMENT);*
3. Set current transaction currency: *AcceptSDK.setCurrency(CurrencyCode.USD.name());* because Alipay is supported only for USD
4. Fill up payment item:, *AcceptSDK.addPaymentItem()* – it adds a new item to customer's basket. The final amount is calculated from basket's content.
5. Scan QR code and set his content into SDK using method: AcceptSDK.setAlipayCustomerCode(String custommerCode);
6. And call on SDK *AcceptSDK.postAlipayPayment (new OnRequestFinishedListener<Payment>())*

- GitHub demo example of Alipay activity

4.7 Cash payment implementation

Cash Payment is little bit different by the implementation as normal payment using terminal, partially similar to Alipay.. Cash payment is not using terminal. We can specify it as just sending information to Accept backend about some income made by cash. On backend side you will have then record of successful cash transaction. It can be implemented by few simple calls.

1. *AccpetSDK.startPayment()* – it clears the data and starts a fresh payment.
2. Set SDK payment mode to Cash by simple: *AcceptSDK.setPaymentTransactionType(AcceptSDK.TransactionType.CASH_PAYMENT);*
3. Fill up payment item:, *AcceptSDK.addPaymentItem()* – it adds a new item to customer's basket. The final amount is calculated from basket's content.
4. And call on SDK *AcceptSDK.postCashPayment(new OnRequestFinishedListener<Payment>())*

- GitHub demo example of cash activity

4.8 PaymentFlowDelegate

Is basic representation of interface used for whole payment process.
Simple explanation and example:

```
paymentFlowController.startPaymentFlow(device, amountUnits, amountCurrency,
        new PaymentFlowController.PaymentFlowDelegate() {

    @Override
    public void onPaymentFlowUpdate(PaymentFlowController.Update update) {
        // Callback notifies about the progress of payment.
    }

    @Override
    public void onPaymentFlowError(PaymentFlowController.Error error,
                                   String technicalDetails) {
        // Callback notifies about an error occurred during the payment.
    }

    @Override
    public void onPaymentSuccessful(Payment payment, String TC) {
        // Callback notifies about a successful end of the payment.
    }

    @Override
    public void onSignatureRequested(PaymentFlowController.SignatureRequest
signatureRequest) {
        // Callback delegates the payment signing to the application.
        // Customer needs to draw a signature and then application needs to call
signatureEntered() or signatureCanceled().
    }

    @Override
    public void onSignatureConfirmationRequested(
            PaymentFlowController.SignatureConfirmationRequest signatureConfirmationRequest)
{
        // Callback delegates signature confirmation to the application.
        // The app needs to inform merchant he needs to confirm the signature using
terminal's keypad.Some terminals are handling this feature using terminal display and
buttons.
    }
});
```

4.8.1 onPaymentFlowUpdate

callback method informs about what is happening with the SDK. The method can be called very often comparing to the others. Usually, the implementation of the method should update a label informing merchant and customer about the state of payment. The list of updates is available at **Appendix A.**

4.8.2 onPaymentFlowError

callback method is invoked when an error occurred during the payment. The list of errors is available at **Appendix B.** After this callback no more methods will be called. Proper implementation should after thi call *cancelPaymentFlow()* method from payment flow controller

4.8.3 onPaymentSuccessful

callback method is invoked when the payment is accepted. After this callback no more methods will be called. Proper implementation should after thi call *cancelPaymentFlow()* method from payment flow controller.

4.8.4 onSignatureRequested

callback method is invoked when payment procedure requires customer's signature. The app should display a drawing canvas and collect the signature image file from what user drawn. When the signature is ready the SDK is notified by **SignatureRequest:: signatureEntered(byte[] pngBytes).** This callback is optional depending which terminal and card are used.

4.8.5 onSignatureConfirmationRequested

callback method is invoked when the SDK asks to display a message to a merchant that he needs to confirm signature on terminal's keypad. The application should display a signature drawn by the customer and ask merchant to confirm it using terminal's keypad. This callback is optional depending which terminal and card are used.

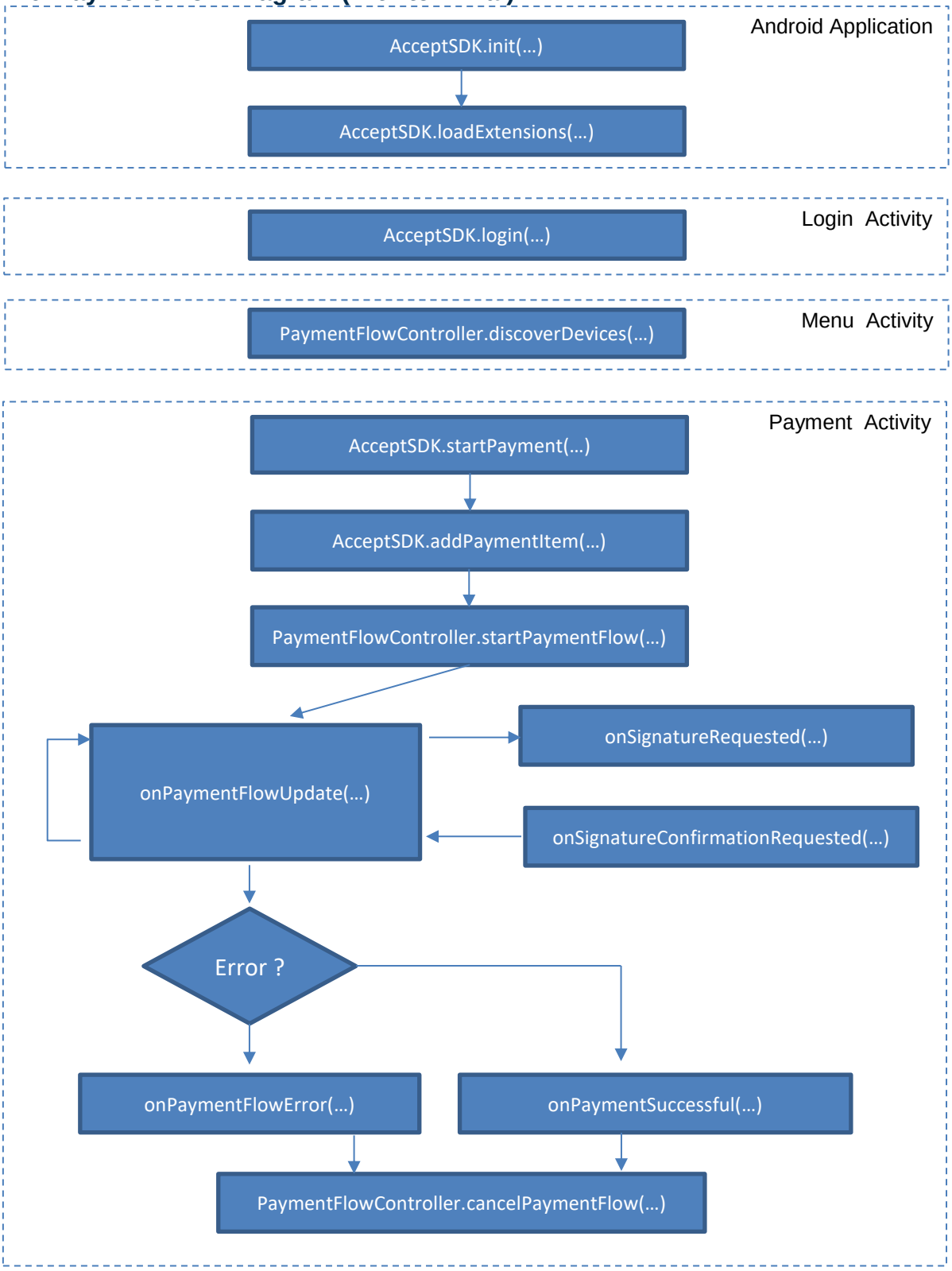
- **Note:** all implementations example you can find on GitHub demo example

4.9 Canceling payment flow

Abstract PaymentFlowController contains very important method *cancelPaymentFlow().*
This method is native SDK support for clean up all necessary after payment remembered parameters, reset all attributes to default, in case of Spire extension this method is also responsible for close necessary connection automatically to terminal and switch him to stand by mode. Terminal will stay in state prepared for next transaction.

- **Note:** implementations example you can find on GitHub demo cancelPaymentFlow

4.10 Payment Flow Diagram (with terminal)



5 Additional features

This part of document will mention about how to implement step after step all additional features.

5.1 Transaction history

Transaction history is basic part of features supported by SDK. By the simple implementation of one interface and method, you have access to basic functionality :

```
public interface OnRequestFinishedListener<T> {
    void onRequestFinished(ApiResult var1, T var2);
}

public static boolean getPaymentsList(int pageNumber, int pageSize, Long since,
Long until, OnRequestFinishedListener<List<Payment>> listener)
```

How to use this feature you can find in demo app example. You have to just specify some basic attributes as number of page, size of one page, some dates.
This request support pagination, it means you can specify if you need ”third pade of size 20 rows with specified date from/ to filtration” or for display last few transactions is it just “first page of 10 transactions with unspecified time filtering attributes”. Fist setup will send you filtered data by the date, orderd by the time and data index from 41 to 60 (because mentioned third page and page size is 20).Second setup of conditions will respond with just 10 latest transactions.

- **Note:** implementations example of history data request you can find on GitHub demo history data

5.2 Reverse and refund transaction

This feature is more sophisticated. It can be blocked by configuration on backend side. Some integrations are simply not using this feature. To use this feature pls contact our management a discuss usage. Everything important related to this feature is explained in one AsyncTask.

```
public class ReverseOrRefundAsyncTask extends AsyncTask<Void, Void,
AcceptBackendService.Response> {

    private final Payment payment;

    public ReverseOrRefundAsyncTask(Payment payment) {
        this.payment = payment;
    }

    @Override
    protected AcceptBackendService.Response doInBackground(Void... params) {
        if(payment.isReversible())
            return AcceptSDK.reverseTransaction(payment.getTransactionId());
        else if(payment.isRefundable())
            return AcceptSDK.refundTransaction(payment.getTransactionId());
        return null;
    }

    @Override
    protected void onPostExecute(AcceptBackendService.Response response) {
        if(response != null) {
            if (response.hasError()) {
                Toast.makeText(getApplicationContext(),
response.getError().toString(), Toast.LENGTH_LONG).show();
            }
```

```
            else {
                AcceptTransaction body = (AcceptTransaction) response.getBody();
                if(body.status == AcceptTransaction.Status.reversed) {
                    Toast.makeText(getApplicationContext(), "Transaction was
reversed", Toast.LENGTH_LONG).show();
                    payment.setStatusToReversed();
                    ((PaymentAdapter)
listView.getAdapter()).notifyDataSetChanged();
                }
                else if(body.status == AcceptTransaction.Status.refunded) {
                    Toast.makeText(getApplicationContext(), "Transaction was
refunded", Toast.LENGTH_LONG).show();
                    payment.setStatusToRefunded();
                    ((PaymentAdapter)
listView.getAdapter()).notifyDataSetChanged();
                }
                else
                    Toast.makeText(getApplicationContext(), "Transaction was
reversed or refunded", Toast.LENGTH_LONG).show();
            }
        }
        else
            Toast.makeText(getApplicationContext(), "Can not be reversed or
refunded", Toast.LENGTH_LONG).show();

        loading.setVisibility(View.GONE);
    }
}
```

5.3 Receipt builder

Additional Accept SDK support, for creating receipt is covered by ReceiptBuilder.class. This builder contains all important getters for values which are usually displayed on receipt. Structure and design of receipt is left to implementer.

- **Note:** implementations example of receipt you can find on GitHub demo receipt example

5.4 Payment additional attributes

Additional Accept SDK support, for some special parameters is as example availabele in AdditionalPaymentFieldActivity.class. Usage of this parameters is strongly recomended to consult with management before.
Simply set this attributed directly into SDK using setters, and during payment if attribute is filled out, it will be send together with next payment. Method cancelPayment is cleaning this attributes.

- **Note:** implementations example of additional params you can find on GitHub demo example

5.5 Payment sub merchant information attributes

Sometimes it is required to make some merchant hierarchy. For this case SDK supports to add additional sub merchant information, represented by the class SubMerchantInfo.class.
Use please simple setter : *AcceptSDK.setSubMerchant(subMerchantInfo)* , and again if attribute is filled out, it will be send together with payment. Method cancelPayment is cleaning this attributes.

- **Note:** implementations example of submerchant info you can find on GitHub demo example

5.6 Payment additional “Usage” attribute

Sometimes a note can also be attached to transaction which shows up on bank statement of the customer. This is helpful in referencing a transaction. For this case SDK supports to add “usage” attribute (max 256 characters). To use this simply call: AcceptSDK.setUsage(String usage). If attribute is filled out, it will be send together with payment. To display this value (or check if its set), simply call AcceptSDK.getUsage().

This feature is not supported by all the acquirers. The size of this field depends on the acquirer.

6 Appendix: A

Payment flow updates class details (**PaymentFlowController.Update**):

CONFIGURATION_UPDATE	SDK updates terminal's configuration remotely.
CONFIGURATION_UPDATE_AVAILABLE	SDK notifies about new configuration update is available.
FIRMWARE_UPDATE	SDK updates terminal's firmware remotely.
FIRMWARE_UPDATE_AVAILABLE	SDK notifies about new firmware update is available
LOADING	SDK is processing please wait.
COMMUNICATION_LAYER_ENABLING	SDK is processing please wait.(waiting for some communication delays)
RESTARTING	SDK restart the terminal an waits for the reboot.
ONLINE_DATA_PROCESSING	SDK performs online processing of the payment.
EMV_CONFIGURATION_LOAD	SDK load properties required for CHIP transactions.
DATA_PROCESSING	SDK notifies about the communication between the card and terminal.
WAITING_FOR_CARD_REMOVE	SDK get a information from the terminal it waits for merchant to remove the card. An application should display a text: "PLEASE REMOVE CARD".
WAITING_FOR_INSERT	SDK get a information from the terminal it waits for merchant to insert the card. An application should display a text: "PLEASE INSERT CARD".(force to insert)
WAITING_FOR_INSERT_OR_SWIPE	SDK get a information from the terminal it waits for merchant to insert or swipe the card. An application should display a text: "PLEASE INSERT OR SWIPE CARD". (if contact less not enabled)
WAITING_FOR_INSERT_SWIPE_OR_TAP	SDK get a information from the terminal it waits for merchant to insert, swipe or tap the card. An application should display a text: "PLEASE PRESENT OR INSERT". (default)
WAITING_FOR_PINT_ENTRY	SDK get a information from the terminal it waits for the PIN entry. An application should display a text: "PLEASE ENTER PIN".
WAITING_FOR_SWIPE	SDK get a information from the terminal it waits for merchant to swipe the card. An application should display a text: "PLEASE SWIPE CARD". (force to swipe)
WAITING_FOR_AMOUNT_CONFIRMATION	SDK get a information from the terminal it waits for customer to confirm the amount. An application should display a text: "PLEASE CONFIRM AMOUNT".

TRANSACTION_UPDATE	SDK performs another online processing of the transaction to finally commit it. (last part of transaction processing)
WRONG SWIPE	SDK inform about wrong swipe
TERMINATING	SDK inform about termination if some task
DONE	SDK inform about finalized configuration or firmware update
UNKNOWN	SDK experienced an state which is not handled yet.

- GitHub demo example of implementtion

7 Appendix: B

Payment error class details (**PaymentFlowController.Error**):

Note: Not all erros can occur while using particular extension. For example, Audio Jack terminals will never report **NO_BLUETOOTH_MODULE** error.

COMMUNICATION_FAILED	An IO error occurred during data transmission to or from the terminal.
PAIRING_LOST	The Bluetooth device is not paired anymore.
DATA_PROCESSING_ERROR	Transaction processing by the card has failed.
ONLINE_PROCESSING_FAILED	Transaction rejected by backend.
ONLINE_PROCESSING_FAILED _CONNECTION_PROBLEM	No internet connection.
TRANSACTION_UPDATE_FAILED	Transaction finalization failed.
TRANSACTION_UPDATE_FAILED _CONNECTION_PROBLEM	No internet connection.
EMV_CONFIGURATION_LOAD_FAILED	Failed to load the EMV configuration.
EMV_CONFIGURATION_LOAD_FAILED _CONNECTION_PROBLEM	No internet connection.
EMV_CONFIGURATION_INVALID	The validation of EMV configuration failed.
CANCELED_ON_TERMINAL	The payment was cancelled using a button from terminals' keypad.
UNKNOWN	SDK experienced an error which is not handled yet.