wirecard

Quick Start Guide

Android version

Author: Wirecard Technologies GmbH

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

Accept SDK Android Documentation - Full manual for Android SDK.

1.3 Revision History

Sample App Version	Date	Name	Comments
1.0.0	06.08.2015	Damian K.	Fixsed typos
1.0.0	31.07.2015	Damian K.	Initial version.

Copyright

Trademarks

1. 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 (Thyron) Posmate
2. IDTech Unimag
3. BBPOS EMVSwiper

Accept SDK for Android is distributed as a set of AAR libraries. 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-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 Spire (Thyron) terminals for the payments, the SDK should have following aar files:

- accept-sdk-android-acceptsdksource-x.x.x.aar
- accept-sdk-android-thyronextension-x.x.x.aar

2. SDK Integration

To integrate the Accept SDK please follow the steps below:

1. Copy Accept SDK AARS files into your project directory and modify **build.grade** file:

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

repositories {
    mavenCentral()
    flatDir {
        dirs 'aars'
    }
}

buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:1.2.3'
    }
}

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'
    }
}

dependencies {
    compile (name: 'accept-sdk-android-acceptsdksource-1.4.7', ext: 'aar')
    compile (name: 'accept-sdk-android-thyronextension-1.4.7', ext: 'aar')
}
```

2. 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. Add custom **Application** class to your project. If it's already there, skip this step, if not, please create a new Java class App.java and add it to the project:

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

    @Override
    public void onCreate() {
        super.onCreate();
    }
}
```

Then, modify **AndroidManifest.xml** file to use the class:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android">
    ...
    <application android:name=".App">
        ...
    </application>
    ...
</manifest>
```

4. Add Accept initialisation procedure to Application class:

```
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 ClientID/Secret from Accept support team.

5. Add Accept's configuration properties to **strings.xml** resources:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
     ...
    <item name="acceptsdk_debug" type="bool">true</item>
    <item name="acceptsdk_certification_mode" type="bool">true</item>
    <item name="acceptsdk_swiper_type" type="string"/>
    <item name="acceptsdk_terminal_type" type="string">ThyronExtension</item>
    <string-array name="extensions_list">
        <item>ThyronExtension</item>
    </string-array>
    ...
</resources>
```

Note: The configuration properties are responsible for selecting the extension used by the SDK. In the example above the SDK will try to load an extension called "ThyronExtenion". Please obtain your configuration from Accept support team, if it's not delivered together with the bundle.

3. Payments Integration

The payment procedure starts from a login operation. It initialises 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());
    }
});
```

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();
```

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**.

PaymentFlowController exposes two important methods:

1. **Devices discover method**. A query method to get the list of avaible devices. For example, for bluetooth enabled terminals, the method will get bluetooth paired devices which fit the requirments.

```
void discoverDevices(Context context, DiscoverDelegate delegate);
```

2. **Payment lanuch method**. The entry point for starting the payment.

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

3.1 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 it will return the list of supported terminals. A sample code for device discovery is presented below:

```
public class MyActvity extends Activity {

  private final PaymentFlowController paymentFlowController =
      new AcceptThyronPaymentFlowController();

  @Override
  public void onCreate(Bundle savedInstanceState, PersistableBundle persistentState) {
    super.onCreate(savedInstanceState, persistentState);
    paymentFlowController.discoverDevices(MyActvity.this,
        new PaymentFlowController.DiscoverDelegate() {

      @Override
      public void onDiscoveryError(PaymentFlowController.DiscoveryError error,
                                   String technicalMessage) {
        runOnUiThread(new Runnable() {
          @Override
          public void run() {
            // Display an error.
          }
        });
      }

      @Override
      public void onDiscoveredDevices(List<PaymentFlowController.Device> devices) {
        runOnUiThread(new Runnable() {
          @Override
          public void run() {
            // Show a picker to select device.
          }
        });
      }
    });
  }
}
```

3.2 Handling payment

The payment procedure is driven by the SDK. It 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 **PaymentFlowDelegate** interface is presented below:

```
AcceptSDK.startPayment();
AcceptSDK.addPaymentItem(
    new PaymentItem(1, "Sample Basket Item", new BigDecimal(10.00), 0));

final Currency amountCurrency = Currency.getInstance(AcceptSDK.getDisplayCurrency());
final long amountUnits =
        AcceptSDK.getPaymentTotalAmount().longValue() * (int) Math.pow(10, amountCurrency.getDefaultFractionDigits());

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.
    }
});
```

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. paymentFlowController.startPaymentFlow() – it runs the payment, establishes a session with the terminal and goes through the payment.

After (3) the instance of **PaymentFlowDelegate** interface is used for:

onPaymentFlowUpdate callback method informs about what's 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.**

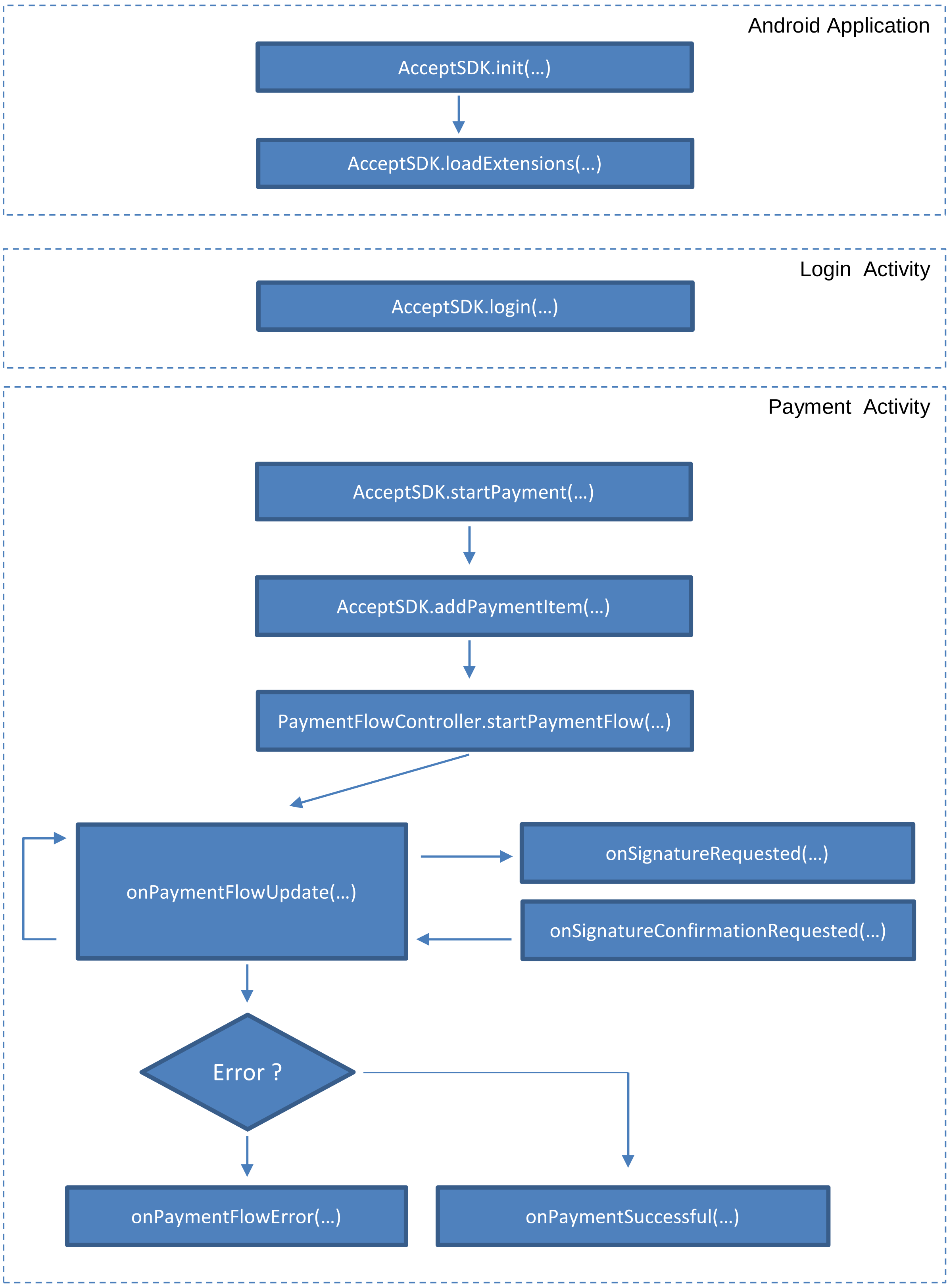
onPaymentFlowError callback method is invoked when an error occurred during the payment. The list of errors is available at **Appendix B.** After this callback no more methods will be called.

onPaymentSuccessful callback method is invoked when the payment is accepted. After this callback no more methods will be called.

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.

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.

3.3 Payment Flow Diagram


Android Application
AcceptSDK.init(...)
AcceptSDK.loadExtensions(...)
Login Activity
AcceptSDK.login(...)
Payment Activity
AcceptSDK.startPayment(...)
AcceptSDK.addPaymentItem(...)
PaymentFlowController.startPaymentFlow(...)
onPaymentFlowUpdate(...)
onSignatureRequested(...)
onSignatureConfirmationRequested(...)
Error ?
onPaymentFlowError(...)
onPaymentSuccessful(...)

4. Appendix: A

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

CONFIGURATION_UPDATE	SDK updates terminal's configuration remotely.
FIRMWARE_UPDATE	SDK updates terminal's firmware remotely.
RESTARTING	SDK restart the terminal an waits for the reboot.
ONLINE_DATA_PROCESSING	SDK performs online processing of the payment.
ONLINE_DATA_PROCESSING_FINISHED	SDK finished the online processing.
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".
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".
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".
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.

5. 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.

NO_BLUETOOTH_MODULE	The device is not equipped with Bluetooth module.
NO_PERMISSION_TO_USE_BLUETOOTH	The application does not have rights to use the Bluetooth module. No Bluetooth permissions in AndroidManifest.xml file.
BLUETOOTH_COMMUNICATION_FAILED	An IO error occurred during data transmission to or from the terminal.
BLUETOOTH_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.