wirecard

Accept SDK

Demo & Integration Guide
iOS Version 1.6.200

Table of Contents

0. Revision History

Rev	Date	Author	Description
			Please refer to CHANGELOG.md

1. Overview

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

Currently the following terminals are supported:
- Spire Posmate,
- IDTech Unimag,
- BBPOS EMVSwiper,
as well as Apple Airprint compatible printers and Datecs DPP250 printer.

The Accept SDK is delivered together with a Demo App for training and example purposes. The screenshots in this document come from this app.

Note: Do not forget to set your backend URL, client ID and client secret in the initialization of *Utils* instance (more details in next sections) before running the app. Also you should have a user account created in the backend before loging in.

1.1 System Requirement

Target System:

Accept SDK can only run on an armv7 compatible iOS device (iPhone 3GS or above), and requires iOS 7 or above. Please note that iOS8.1 or above is highly recommended due to Apple Bluetooth bugs.

1.2 Xcode Project Settings

Accept SDK needs several frameworks to be included into your Xcode project, together with the libPodsSDK library that is part of the SDK. The list of mandatory frameworks is included below:

Note: Accept SDK includes zlib library so do not include it again in your project to avoid duplicate symbols.

▼ **Linked Frameworks and Libraries**

Name	Status
MediaPlayer.framework	Required ⬍
libstdc++.dylib	Required ⬍
CoreData.framework	Required ⬍
libz.dylib	Required ⬍
libstdc++.6.dylib	Required ⬍
AddressBook.framework	Required ⬍
libsqlite3.dylib	Required ⬍
libxml2.dylib	Required ⬍
libxml2.2.dylib	Required ⬍
UIKit.framework	Required ⬍
Foundation.framework	Required ⬍
AudioToolbox.framework	Required ⬍
AVFoundation.framework	Required ⬍
CFNetwork.framework	Required ⬍
CoreAudio.framework	Required ⬍
CoreGraphics.framework	Required ⬍
libPodsSDK.a	Required ⬍
MobileCoreServices.framework	Required ⬍
Security.framework	Required ⬍
SystemConfiguration.framework	Required ⬍
ExternalAccessory.framework	Required ⬍
QuartzCore.framework	Required ⬍
CoreBluetooth.framework	Required ⬍
CoreText.framework	Required ⬍

\+ −

Note: Do not forget to include the flag –ObjC in your *Build Settings -> Other Linker Flags* when importing the Accept SDK Library.

Note: For proper use of our backend certificates, your info plist file will have to set the transport security setting as in the screenshot:

▼ App Transport Security Settings	⬍	Dictionary	(1 item)
Allow Arbitrary Loads	⬍	Boolean	YES

1.2.2 Activating Developer Mode

Defining the flag *DEV_MODE_ENABLED=1* in your project's Preprocessor Macros, and setting *UIFileSharingEnabled (Application Supports iTunes file sharing)* key in the info plist of your app, will allow the creation of log files in your app's document folder every time a transaction is executed (successfully or not).

For accessing to these log files, connect your apple device to a computer with iTunes installed, go to Apps section, selection your app, and import the text file (or files) you find there. Depending on the terminal you were using, the filename will change (i.e. BBPOS/SpirePosmate, etc.).

Note that only non-critical content will be offered unencrypted.

DEV_MODE_ENABLED=1 will also print an extra set of debug logs, including online requests (marked with blue icons) and responses (green icons). Error will appear as red icons.

1.3 Background Modes

The application uses the following background modes. These are enabled to improve the stability of the application when moving to background mode

Required background modes			Array		(4 items)
Item 0			String		App downloads content from the network
Item 1			String		App communicates with an accessory
Item 2			String		App shares data using CoreBluetooth
Item 3			String		App communicates using CoreBluetooth

Note: During the AppStore submission process it is important to notify the Apple review officer and explain why you include the above background modes. These background modes allow extra seconds of execution and increase the stability of the communication between the app and the terminal in the payment flow. Please include the above explanation in the review notes for the Apple Review officer in order to avoid the app rejection.

Note: Even though having also Audio Background Mode (apart from the four modes above) in your app would make it more stable (using jack-connected terminals), Apple does not accept it for non-recording audio apps. Trying to add more modes than the specified here risk a rejection in the store.

1.4 Terminals Setup, Protocols and MFI

Terminal Setup

Accept SDK includes a file called *paymentmethods.xml* that should be also included in your own project (within *Resources* folder). This XML file defines the names of the terminals as they are discovered, the blacklisted cards for each terminal, and also defines what terminals are supported by the application.

Note: Take care of keeping the structure of *paymentmethods.xml* when modifying it.

External Accessory Protocol

Your App must inform the operating system what hardware is allowed to establish a safe BT communication. The *Supported External Accessory Protocols* configuration setting defines all protocols that the application will be using when communicating with supported terminals.

In the Demo App, Thyron, IDTech and Datecs protocols are included:

▼ Supported external accessory protocols	◆	Array	(3 items)
Item 0		String	com.valencetech.iBT–02.ibridge
Item 1		String	com.thyron
Item 2		String	com.datecs.printer.escpos

MFI Program

Before submitting your iOS app to iTunes for distribution, Apple requires all iOS apps that communicate with Apple approved MFI devices to be registered with Apple. This registration process officially associates your app with the vendor terminal and can only be registered by the terminal vendor.

When requesting MFI program registration from the vendor, the following info must be part of the requesting email (i.e. for Datecs vendor):

Application name : MyApp
Version : 1.0.1 (version to be released)
Bundle id : com.mycompany.myappname
Device protocol : com.datecs.printer.escpos (see above, External Accessory Protocol)

Vendor contact links

mpos_technical_support@spirepayments.com
http://www.datecs.bg/en/Contact-us/1
http://www.idtechproducts.com/contact.html

1.5 Microphone Access and Audio Issues

Microphone access must be allowed by the user the first time the application is run in order to use the audio jack for devices such as Unimag and EMVSwiper. In the case this setting is off (iPhone/iPad Settings->Privacy->Microphone), the SDK will display the error *AcceptHardwareAccessPermissionErrorCode*. Make sure to convey this message to the user and inform them to modify their settings to allow the microphone access for your application.

Note: Because of the nature of the audio connector, it is not possible to detect what device is inserted. Always force the user to select manually their terminal, as a mismatch between the hardware and the setting can produce unexpected results.

Note: Volume levels could affect the Audio connector signal, even making the initialization impossible. This is a very rare issue in the latest iOS updates, but the issue still exists nevertheless.

1.6 Bluetooth Issues

Bluetooth devices not only need to be paired before use: they need to be free of previous pairing. If some device was used before in another phone or tablet, make sure that it is unpaired. Trying to pair terminal already paired with different device could produce errors ("Device out of range, etc.") and lead to failures in the initialization process.

Note: Bluetooth stability is highly dependent on the operating system. If after upgrading to latest iOS version you suddenly experience Bluetooth issues please check on forums if they are related to Apple BT bugs.

2. The Accept Class

The Accept class is the core of the Accept SDK and it offers all the methods necessary for user actions, terminal communication and backend requests.

Strings IDs for the different compatible terminals

```
extern NSString * const AcceptSpireVendorUUID;
extern NSString * const AcceptbbPOSVendorUUID;
extern NSString * const AcceptIDTECHVendorUUID;
extern NSString * const AcceptDatecsVendorUUID;
extern NSString * const AcceptVeriFoneVendorUUID; //Deprecated
extern NSString * const AcceptErrorDomain;
```

Main Accept class

```
/**
* @class Accept
* @discussion Main SDK class. It contains all public properties and functions
**/
@interface Accept : NSObject

///version Current SDK version
@property (nonatomic, readonly) NSString * version;
```

2.1 Prior steps

The Accept Demo App uses a shared instance of a class called *Utils* for an easy access to basic settings, such as backend configuration and user token. The first time the *Utils* instance is allocated, a backend URL and a clientId/clientSecret are set. The client values are mandatory and should be provided by the backend administrator. These values are used for authentication, so they should be kept secret.

Also notice that the currency and country set in the backend have to match the ones in the app. (Only) if you have problems with declined transactions, ask us the backend setting and then edit, in the file Utils.h, the constant CURRENCY_CODE_DEFAULT_VALUE.

3. User Actions and Payments

How to Login

The first action the user should do is login in to the system, as this will provide a session token that allows more specific actions. This token should be stored and used until it expires. Then it must be manually refreshed by loging in again. (The Demo app sets an internal timer of 5 minutes, but in real life this will depend on the backend settings).

Note: Tokens in backend side are kept alive for the time the user is performing requests and transactions, and it expires after a period of inactivity. For usability reason, it is recommended to show the login screen automatically after the expiration, and to save the payment basket so the user can recover the unfinished transaction later.

In the screenshot above it is shown the Demo App's first view, the Login screen. The segmented controller under the *token info* label allows the selection of the proper Login request, the request to recover the User ID, the request to reset the password, or the request to change the current password. These options are executed by the following Accept methods:

```
/**
 * @brief Request the user session access token (that will expire if no activity is
executed after several minutes)
 * @param username unique username for login in
 * @param password user password
 * @param config Instance needed to use backend services
 * @param completion Block that receives the server response or an error
 **/
- (void) requestAccessToken:(NSString*)username
          password:(NSString*)password
            config:(AcceptDataServiceConfig*)config
        completion:(void (^)(AcceptAccessToken*, NSError*))completion;
```

```
/**
 * @class AcceptAccessToken
 * @discussion Access Token info class. access token and its expiration time
 **/
@interface AcceptAccessToken : NSObject
@property (nonatomic, strong) NSString * accessToken;
@property (nonatomic) NSNumber * expireInSeconds;
@end
```

requestAccessToken:password:config:completion accepts the username and password as strings, the backend configuration previously defined, and a completion block that in our case saves the response data and the terminal configuration. Accept Demo app, as can be seen in the code, is a very simple example, but in a real world scenario, the completion block would be also saving info such as tax rates, currencies available, amount limits, etc.

```
/**
 * @brief Request merchant info from an user session token
 * @param accessToken The user session token for authentication
 * @param config Instance needed to use backend services
 * @param completion Block that receives the merchant info or an error
 **/
- (void) requestMerchantInfo:(NSString *)accessToken
            config:(AcceptDataServiceConfig*)config
        completion:(void (^)(AcceptUserResponse *, NSError *))completion;
```

```
/**
 * @brief Request for the config file in backend
 * @param accessToken The user session token for authentication
 * @param config Instance needed to use backend services
 * @param version Current version already downloaded. This can be hardcoded to zero
for simplification.
 * @param completion Block that receives the config file or an error
```

```
 * @return
 **/
- (void) queryConfigFile:(NSString*)accessToken
           config:(AcceptDataServiceConfig*)config
    andCurrentVersion:(NSString*)version
       completion:(void (^)(AcceptTerminalConfigFiles*, NSError*))completion;
```

requestMerchantInfo:accessToken:config:completion and
queryConfigFile:config:andCurrentVersion:completion are functions usually
called once the user token has been retrieved and the merchant info and
configuration files are needed.

forgotUserID, resetPassword, changePasswordForToken are three user-
related methods that require different parameters but are handled in a similar
way:

```
/**
 * @brief Request for email with the userId on it
 * @param email email address to receive the content
 * @param config Instance needed to use backend services
 * @param completion Block that will receive a boolean indicating the success or an
error
 **/
- (void) forgotUserId:(NSString *)email
         config:(AcceptDataServiceConfig*)config
      completion:(void (^)(BOOL, NSError*))completion;
```

forgetUserID:config:completion accepts email address as string, for the
backend to send the UserID reminder to. This email address should be the same
as the user's defined email address in backend. This method also requires the
backend configuration and a completion block to handle the response.

```
/**
 * @brief Resetting the password of a user
 * @param userId the unique user id
 * @param config Instance needed to use backend services
 * @param completion Block that will receive a boolean indicating the success or an
error
 **/
- (void) resetPassword:(NSString*)userId
         config:(AcceptDataServiceConfig*)config
      completion:(void (^)(BOOL, NSError*))completion;
```

resetPassword:config:completion accepts the userID as a string. The new
password will be send to the user email address as set in the backend. Also
requires the backend configuration and completion block to handle the response.

```
/**
```

```
 *  @brief Request for changing the password
 *  @param userToken The user session token for authentication
 *  @param newPassword The new password following the specs of the backend
 *  @param reNewPassword Confirmation for the password (second input of the same
string to be accepted)
 *  @param config Instance needed to use backend services
 *  @param completion Block that will receive a boolean indicating the success or an
error
 **/
- (void) changePasswordForToken:(NSString*)userToken
            newPassword:(NSString*)newPassword
         reNewPassword:(NSString*)reNewPassword
              config:(AcceptDataServiceConfig*)config
           completion:(void (^)(BOOL, NSError*))completion;
```

changePasswordForToken:newPassword:reNewPassword:config:completion
must receive a bigger set of parameters, that includes the user's token, the new
one the user has chosen, as well as a confirmation of the new password. It also
receives, as usual in these methods, the configuration of the backend and a
completion block to handle the response.

Note*: For security reasons, it is recommended to ask for the current password
and do the change password action in a new requested token. The nesting of
requesting token + changing password is implemented as example in the Demo
App for you to check.

3.1 Discovering and updating hardware

No SIM 📶   12:37   🔋⚡

2000

BBPOS | uEMVSwipe

EUR - €

Pay   Revert

Valid token

Datecs | -

Print

Progress will be shown here

⚫ Login   🔵 Payment   ⚫ Request

The second screen in the Demo App is the Payment view, which shows the amount entry (note that all amounts appear as multiplied by 100; this will be explained in the section *Creating and executing a transaction*); the terminal selector; the currency selector (though the Demo App has euro hardcoded); a button for paying; a button for reversing an already approved and executed payment; a printer selector; a button for printer; and two labels: one with the status of the token, and a second at the bottom with the progress of the payment flow.

The task of populating the UI with the available terminals, printers, etc. is executed by the methods:

```
/**
 * @brief Adquire an array of available terminal vendors
 * @param completion The array of vendors found, returned with objects of class
AcceptTerminalVendor
 **/
- (void) discoverSupportedVendors:(void (^)(NSArray *))completion;

/**
 * @brief Adquire an array of terminal models for a single vendor
```

```
 *  @param vendorUUID The Unique vendor id string. vendorUUIDs are adquired using
discoverSupportedVendors
 *  @param completionBlock Block that receives the array of AcceptTerminal objects or
an error
 **/
- (void) discoverTerminalsForVendor:(NSString*)vendorUUID
                completion:(void (^)(NSArray *, NSError*))completionBlock;


/**
 *  @brief Discover supported printer vendors
 *  @param completion Block that will receive an array of AcceptPrinterVendor objects
 **/
-(void) discoverSupportedPrinterVendors:(void (^)(NSArray *))completion;


/**
 *  @brief Discover the printer types for a brand/vendor
 *  @param vendorUUID The Unique vendor id string. vendorUUIDs are adquired using
discoverSupportedPrinterVendors
 *  @param completion Block that will receive an array of AcceptPrinter objects and an
error if it exists
 **/
- (void) discoverPrintersForVendor:(NSString*)vendorUUID
                completion:(void (^)(NSArray *, NSError*))completion;
```

Once a terminal has been selected, there is a check of the configuration file
version installed on the device. This operation (only necessary by EMV capable
terminals such as Posmate), is done by the method *updateTerminalsForVendor*. If
the version available in the server is newer than the installed one (or the
terminal was never updated), a configuration zip file will be downloaded to the
app, unzipped and transferred to the device.
As a developer, it is your responsibility to decide when an update is required or
not. The Demo app shows a simple version tracking as examplethe unique
Posmate name combine with NSUserDefaults.

The first time the app runs -> nothing in user default was set, thus a
configuration update is required.
If the number provided by getVersion is bigger than the saved -> configuration
update is required.
Else, ignore.

If for some hypothetical reason the Posmate name changes through firmware
(and the firmware can be only updated through the backend+SDK) ->
configuration update is required, as firmware update resets everything.

```
/**
 *  @brief Update terminals for a vendor. Chip compatible terminals receive
config files from backend; this function will decide if the updates are needed
according current version
 *  @param terminal The terminal obtained using discoverTerminalsForVendor
 *  @param vendor The vendor obtained using discoverSupportedVendors
 *  @param token The user session token for authentication
 *  @param config Instance needed to use backend services – will use backend
config from accept.conf if nil
 *  @param progress Progress of the config file update
 *  @param completion Block that receives the update status or error
 **/
- (void) updateTerminal:(AcceptTerminal * _Nonnull)terminal
               vendor:(AcceptTerminalVendor*_Nonnull)vendor
                token:(NSString*_Nonnull)token
               config:(AcceptDataServiceConfig*)config
             progress:(void (^)(AcceptConfigFilesProgress))progress
           completion:(void (^)(AcceptConfigFilesStatus status, NSError*
error))completion;
```

Some terminals (i.e. BBPOS EMVSwiper) have no screen to show the battery
levels. Though they have a critical "very low battery" status that stops the
payment flow with the consequent error, it is convenient to inform in the UI of
your app that the battery is getting too empty. The following function sends the
ID of the vendor of the terminal we are requesting the battery level, and returns
an integer in its completionBlock, which is the percentage of battery remaining.
NOTE: never check for the battery level of a terminal if the terminal is processing
a payment, because the payment will be interrupted.

```
/**
 * @brief Request the battery level of the connected terminal. NOTE: this function
should NOT be called if an operation in the terminal is ongoing (ie startPay has not
finished)
 * @param vendorID Unique id of the terminal's vendor we are requesting the info
 * @param completionBlock Block that will receive the battery level as an integer. If
negative, then it indicates an error between three possible cases: "-1" for showing that
the selected terminalID is wrong or that the terminal does not support battery level info.
"-2" for indicating that the terminal does not respond (not ready or not connected). "-3"
for indicating that the terminal is currently charging through a cable.
 **/
- (void)percentageBatteryRemainingForTerminal:(NSString*)vendorID
completion:(void (^)(NSInteger))completionBlock;
```

```
/**
 * @brief Request the battery level of the connected terminal. NOTE: this function
should NOT be called if an operation in the terminal is ongoing (ie startPay has not
finished)
 * @param vendorID Unique id of the terminal's vendor we are requesting the info
 * @param completionBlock Block that will receive the battery level as a signed integer.
If negative, then it indicates an error between three possible cases: "-1" for showing that
the selected terminalID is wrong or that the terminal does not support battery level info.
"-2" for indicating that the terminal does not respond (not ready or not connected). "-3"
for indicating that the terminal is currently charging through a cable.
 **/
- (void)percentageBatteryRemainingForTerminal:(NSString*)vendorID
completion:(void (^)(NSInteger))completionBlock;
```

3.1.1 Updating firmware

Some terminals (i.e. Spire Posmate, SPm2) allow the remote update of new firmware; this is, the firmware is requested to Accept backend, and the app uploads it to the terminal. This can be done with the functions below.

```
/**
 * @brief Request for the firmware file in backend
 * @param config Instance needed to use backend services
 * @param completion Block that receives the config file or an error
 * @return
 **/
- (void) queryFirmware:(AcceptPaymentConfig*)config completion:(void (^)(BOOL,
AcceptTerminalFirmware*, NSError*))completion;
```

The function *queryFirmware:completion* first ask the backend if there is a firmware available and that version of the firmware is. It is your task, as a developer, to decide if the firmware should be installed.

Installing the firmware can be done with the function:

```
/**
 *  @brief Update terminal's firmware . Chip compatible terminals receive
firmware files from backend; this function will decide if the updates are needed
according current version
 *  @param terminal The terminal obtained using discoverTerminalsForVendor
 *  @param vendor The vendor obtained using discoverSupportedVendors
 *  @param token The user session token for authentication
 *  @param config Instance needed to use backend services
 *  @param progress Progress of the config file update
 *  @param completion Block that receives the update status or error
 **/
- (void) updateTerminalFirmware:(AcceptTerminal * _Nonnull)terminal
                vendor:(AcceptTerminalVendor*_Nonnull)vendor
                 token:(NSString*_Nonnull)token
                config:(AcceptDataServiceConfig*)config
              progress:(void (^)(AcceptConfigFilesProgress))progress
            completion:(void (^)(AcceptConfigFilesStatus status, NSError*
error))completion;
```

```
/**
 * @class AcceptTerminalFirmware
 * @discussion Terminal firmware class. Content required for the Accept terminal
firmware version
 **/
@interface AcceptTerminalFirmware : NSObject
@property (nonatomic, strong) NSString * vendorUUID;
@property (nonatomic, strong) NSString * terminalUUID;
@property (nonatomic, strong) NSString *firmwareUrl;
@property (nonatomic, strong) NSString *firmwareVersion;
@property (nonatomic, strong) NSString *firmwareMD5;
@end
```

The maintenance of new firmware versions will always be handled by Wirecard (backend side). But you, as developer, are responsible for checking if there are new versions available.

Due to the critical nature of firmware updates, you should never force one: firmware updates should always be triggered by users. But you could inform (for example, with a popup) if there is a new one for downloading.

A recommended implementation would be:
- After user logs in, if a terminal is connected, request the terminal details.
- According the terminal details, ask backend for current version of firmware and configuration files. Configuration files can be downloaded automatically (see demo app).
- But if a new firmware is available, prompt a message informing the user about it. The firmware update could be triggered from your settings, once the user feels ready (a stable connection and not-too-low battery level is recommended). The demo app includes a "Firmware update" button for this.

Note: If you are working with production but also test terminals in your development, make sure you use each one with the correct user/merchant. For safety reasons, a test firmware will never be installed in production terminals, and vice-versa.

3.2 Creating and executing a transaction

The main methods for executing a transaction in the demo App are *startPayment*, which provides an example of all callbacks blocks to communicate with the Accept SDK, and *doPaymentThroughVendor*, that prepares the payment configuration and the basket of the transaction.

To identify the processed payment status it is necessary to check AcceptTransaction and Error objects in the completion block of the startPay… methods.

In the case the transaction processing fails on the terminal or due to misconfiguration then the Error will be returned and transaction object may be empty.

In the case the transaction processing is successful the Error will be empty and transaction contains more details.

The payment is **ACCEPTED** if state attribute of the returned transaction equals to "**approved**"
The payment is **NOT ACCEPTED** if the state attribute of the returned transaction equals to one of:
"rejected"
"reversed"

"incomplete"

Note that the example provided in the Demo App is a simplification of real needs. For example, allowGratuity, not used, is a way to provide tips, and your app may need first to enable it as a setting, and then provide the tip value as part of the total amount of the basket. Also notice that there could be a need for different tax values (the Demo App applies no tax at all), and that the basket has a feature for latitude and longitude, not used, in case you want to track GPS location of the device from which the payment originated.

All payments executed by the demo App reproduce the simplest transaction possible: a single item, no charge, no tax.

When presenting the payment view, it was said that all amounts in the Demo are multiplied by 100. The reason is that most currencies have decimal places, generally two of them (i.e. euro and dollar). This multiplication transform an amount of 2000 to the equivalent of 20.00 as backend and payment engines do not work with decimals.

Note: The same is applied to taxes. A tax of 7% is represented by the value 700, as the value 7 represents 0,07%. This multiplied value in the Demo App should be masked in your App UI, though, for usability.

3.2.1 Card Transaction types
There are 4 types of Card transaction the SDK supports at the moment.
Purchase, Authorize, Pre-Authorize and SEPA.
To initiate these types of transaction one can set
`AcceptPaymentConfig.transactionType` attribute to corresponding value.

The Accept method that executes the transaction is *startPay*. Its opposite is cancelPay:

```
/**
 * @brief Start the payment process
 * @param config Instance needed to use backend services
 * @param completion Block that will be called at the very end of payment flow. It
provides an AcceptTransaction object (that may be nil if unauthorised) or a descriptive
error
 * @param progress Block with info to update the UI in base of alerts, errors or general
info messages. Pure feedback for the user
 * @param signatureRequest Block that needs to execute the option to capture a
signature an return the signature data
 * @param signatureVerification Block that informs that a signature needs to be
verified. This is done after the transaction has been already send to backend. Only
terminals with signature verification built it (i.e. Spire Posmate) requires actions here.
signatureVerification block can be handled in the same way as the completion block, as
indicates a completion itself (just lacking the second generate AC)
 * @param appSelection Block informs that an application selection is needed with the
chip card inserted.
 **/
- (void) startPay:(AcceptPaymentConfig*)config
```

```
    completion:(void (^)(AcceptTransaction*, NSError*))completion
      progress:(void (^)(AcceptStateUpdate))progress
      signature:(void (^)(AcceptSignatureRequest*))signatureRequest
signatureVerification:(void
(^)(AcceptTransaction*,AcceptSignatureVerificationResultCallback ,
NSError*))signatureVerification
    appSelection:(void (^)(AcceptAppSelectionRequest *))appSelection;


/**
 * @brief Cancel the payment flow. This is usually called from UI (cancel button when
available) or some error from signature or completion block. Notice that an improper
usage of this function (for example during online communication or level 2 flow in
terminal) can produce unexpected errors
 **/
- (void) cancelPay;
```

The progress of the payment flow and the possible failures are received in the *progress* and *completion* blocks of *startPay*. In the Demo App you can see all available codes and what possible natural sentences could go with them.

The other two callback blocks in *startPay* are *signature* and *appSelection*. The signature block provides an *AcceptSignatureRequest* with the data of an image that represents the customer signature. For simplification, the Demo app uses a dummy picture (signature_sample.png) for this purpose, but your app will need to display a view where the user could sign, and this signature should be compressed to a proper size and send back to the *AcceptSignatureRequest*.

The other block, appSelection, is called from the SDK only when the EMV card inserted has more than one application in its chip. In that case, your app would have to show some kind of picker view, to allow the selection of what app to use (ie. Mastercard or Maestro). For simplification, the demo App automatically selects the first application from the chip (index zero).

Important: If your terminal depends on configuration files (i.e. Spire Posmate/SPm2), you *must* check for updates in backend side and, at least, make sure the user's terminal was configured **at least once**. See section 3.1 for further details.

Note: You also have the option of setting a callback URL for your transaction, meaning that once a payment is executed, there will be a call to the address of your choice. For doing so:
- Check the parameter (AcceptPaymentConfig*)config in *starPay* function.
- Go within its instance *backendConfig*.
- Edit the string variable *callbackURL* on it.

Accept backend will make POST request to callback_url with following parameters:
• unique_id

- guwid
- merchant_id
- user_id
- terminal_id
- terminal_sn
- amount
- currency
- created_at
- merchant_name
- card_brand
- card_number
- state
- error_code
- error_message
- aid_label
- aid
- card_holder

3.2.2 Cash Transaction

To allow the merchant to accept the cash transaction the startCashPayment API can be used to record the transaction in the accept backend.

```
/**
 * @brief Start the Cash payment process
 * @param config Instance needed to use backend services
 * @param completion Block that will be called at the very end of payment flow. It provides an AcceptTransaction object (that may be nil if unauthorised) or a descriptive error
 * @param progress Block with info to update the UI in base of alerts, errors or general info messages. Pure feedback for the user
 **/
- (void) startCashPayment:(AcceptPaymentConfig*)config
        completion:(void (^)(AcceptTransaction*, NSError*))completion
          progress:(void (^)(AcceptStateUpdate))progress;
```

The progress of the payment flow and the possible failures are received in the *progress* and *completion* blocks of *startCashPayment.*

The configuration information does not need to have the terminal information included.

3.2.3 Alipay Transaction

To allow the merchant to accept Alipay transactions `startAlipayPayment` API can be used to initiate the Alipay transaction.

```
/**
 * @brief Start the Alipay payment process
 * @param config Instance needed to use backend services
 * @param completion Block that will be called at the very end of payment flow. It
```

provides an AcceptTransaction object (that may be nil if unauthorised) or a descriptive error
 * @param progress Block with info to update the UI in base of alerts, errors or general info messages. Pure feedback for the user
 **/
- (void) startAlipayPayment:(AcceptPaymentConfig*)config
        completion:(void (^)(AcceptTransaction*, NSError*))completion
         progress:(void (^)(AcceptStateUpdate))progress;

The progress of the payment flow and the possible failures are received in the *progress* and *completion* blocks of *startAlipayPayment.*

The configuration information does not need to have the terminal information included.

The barcode of customers Alipay application needs to be read and the data needs to be set to `alipayConsumerId` parameter in the payment configuration

3.3 Actions on a Payment

The method *requestOperationOnPaymentWithID* allows the execution of three different operations in a payment, from which only one is shown in the Demo: transaction reversals.

The method chosen as an example (*AcceptOperationOnPaymentReverse)*, is the most common one, as *AcceptOperationOnPaymentRefund* is similar, but with an extra cost for the merchant.

Note: A reversal, in a real world scenario, should only be accessible before 24 hours has passed since the payment was approved.

The third operation is a little bit different, *AcceptOperationOnPaymentReceipt*. This operation receives as well the payment unique ID and updates the payment with the email address or phone number to send the receipt to. This is done by sending an instance of the AcceptPaymentParameters class to this API. Either the customers email address or their phone number needs to be set in this instance.

Note: It is important to validate the strings as correct phone numbers and email address before sendint them.

Example for sending the receipt to an email address:
```
AcceptPaymentParameters *paymentParams = [[AcceptPaymentParameters alloc] init];
paymentParams.receiptRequested = true;
paymentParams.customerEmail = @"asd@asd.com";
//paymentParams.customerPhone = @"987654321"; //it checks first if email is defined,
and if not, it sends the receipt through sms if defined
[acceptAPI operationOnPaymentWithID:self.payment.uniqueId
operationMode:AcceptOperationOnPaymentReceipt paymentUpdateParameters:paymentParams
completionBlock:completionBlock];
```

The `AcceptOperationOnPaymentCapture` is used to Capture the previously Authorized or Pre-Authorized transaction.

```
/**
 * @brief Request of one of the available operations: reverse, refund, or prepare a
receipt for a payment
 * @param paymentID Unique id of the transaction to apply the operation
 * @param accessToken The user session token for authentication
 * @param config Instance needed to use backend services
 * @param operation enumerator indicating the operation to be applied
 * @param paymentUpdateParameters Extra paramenters are needed in the case of the
receipt: where to send it, email, phone number, etc
 * @param completion Block that will receive the transaction as the result of the
operation an error if it exists. NOTE: The success of the operation is calculated by
comparing the current status of the transaction to the previous one.
 **/
- (void) requestOperationOnPaymentWithID:(NSString*)paymentID
            andAccessToken:(NSString*)accessToken
                config:(AcceptDataServiceConfig*)config
               operation:(AcceptOperationOnPayment)operation
      paymentUpdateParameters:(AcceptPaymentParameters
*)paymentUpdateParameters
                completion:(void (^)(AcceptTransaction*, NSError*))completion;
```

3.4 Enumerators and Classes

Please check /Documents/index.html

4. Printing Methods

The main method for printing is *startPrint*, and following the same methodology of *startPay*, the printing progression and possible failure handling is done with the respective blocks and methods *printProgress* and *printFailure*.

The SDK allows the printing using Airplay compatible printers as well as Datecs DPP250 printer (iOS compatible ones). In case of problems with the second, please check the battery levels and the Bluetooth pairing.

```
/**
 * @brief Start the printing flow
 * @param config Instance needed to use the printer
 * @param completion Block that will receive the success as a boolean and an error if it exists
 * @param progress Block with the progress status
 * @return
 **/
- (void) startPrint:(AcceptPrinterConfig*)config
      completion:(void (^)(BOOL, NSError*))completion
       progress:(void (^)(AcceptPrinterStateUpdate))progress;
```

Note: For Maestro and Visa certifications, your app's receipts will have to follow certain rules. There are a number of mandatory tags to show, as well as indicator of the card type and transaction type used. And depending if it is contact or contactless, this indicator will have to appear in the header or the section "transaction details". Check out next page's receipt examples.

4.1 Enumerators and classes

Please check /Documents/index.html

Contactless receipt (Mastercard test card)

Customer Receipt

Receipt number 2518
28.01.2016 - 15:47:07
MCC 351 v1 2 CONTACTLESS

SPm2 WDB Test
Einsteinring 35
85609
Munich

Mandatory

10%
No description

| | 1 x | €10,00 | €10,00 |

Prices inclusive of taxes

| Net amount (10 %): | €9,10 | €10,00 |
| Tax 10 %: | €0,90 | |

Total: **€10,00**

Payment details

Payment Date:	28.01.2016 - 15:47:07
Total:	€10,00
Transaction Status:	Approved
Cardholder Name:	a. n.
Card Type:	MCC 351 v1 2
AID:	A0000000041010
MID:	100121112000366
Transaction type:	00
Receipt #:	2518
Approval Code:	00642Z
TC:	
9f36020008	

Payment issued by accept by Wirecard

Contact receipt (Visa test card)

Customer Receipt

Receipt number 2337
26.01.2016 - 15:32:15

SPm2 WDB Test
Einsteinring 35
85609
Munich

10%
No description

| | 1 x | €9,00 | €9,00 |

Prices inclusive of taxes

| Net amount (10 %): | €8,19 | €9,00 |
| Tax 10 %: | €0,81 | |

Total: **€9,00**

Payment details **Mandatory**

Payment Date:	26.01.2016 - 15:32:15
Total:	€9,00
Transaction Status:	Approved
Card #:	XXXX-XXXX-XXXX-0465
	Chip
Cardholder Name:	VISA ACQUIRER TEST CARD 28
Card Type:	VISA CREDIT
AID:	a0000000031010
Terminal ID:	a0000822
MID:	100121112000366
Transaction type:	00
Receipt #:	2337
Approval Code:	153216
TC:	
9f26085e94616bbbb7822b9f2701409f3602000a	

Payment issued by accept by Wirecard

5. Search Methods

The method for receiving existing transactions data is *queryTransactions:config:query:completion*, that receives the already explained token, configuration and completion. What is new is the *AcceptTransactionsQuery* instance sent as parameter that sets the page, page size, order-by method, status and search strings for the query.

As seen in the screenshot above, a text field will help us to search for a date, an amount (without decimals), an ID or even a customer name, while the second segmented control indicates what filter is being applied (regarding the status of the transaction: approved, rejected, reversed or all)

Note: Refunded status is not used in this Demo, but could be used as a fifth filter.

```
/**
 * @brief Search for transactions from a specify query
 * @param accessToken The user session token for authentication
 * @param config Instance needed to use backend services
 * @param query Query for the search
 * @param completion Block that will receive the array of transaction objects (type
AcceptTransaction) or an error
**/
```

```
- (void) queryTransactions:(NSString*)accessToken
        config:(AcceptDataServiceConfig*)config
         query:(AcceptTransactionsQuery*)query
    completion:(void (^)(NSArray*, NSError*))completion;
```

5.1 Transaction Receipt Data

The completion from the function *queryTransactions* above provides an array with transactions. The receipt data can for each individual transaction is accessible through the function *getAcceptReceiptData* of AcceptTransaction. The result is an AcceptReceiptData instance with all content you would need in your receipt.

5.2 Enumerators and classes

Please check /Documents/index.html

6. Statistics Methods
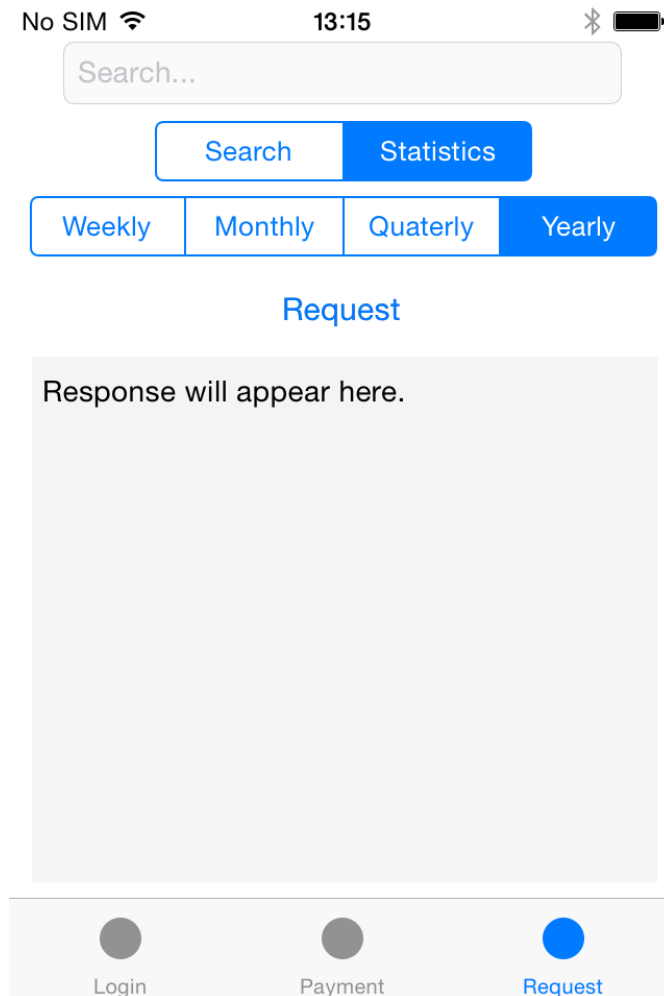
Requesting statistics allows a wide set of options which include nesting (subcategories according the transactions status) as well as group-by periods of time (Daily, monthly, quarterly and yearly).

The method for statistic request is
queryStatistics:config:from:groupBy:statusNesting:completion, that together with the common token, configuration and completion block, accepts the following parameters: the starting and end dates of the interval we want, the groupBy option referring to time measure, an array with the type-grouping (referring to the statistics method: max, min, average, etc) and the Boolean for setting nesting (subcategory by status).

As can be seen in the Demo app, a query statistics call will look like:
```
[self.accept queryStatistics:
[[Utils sharedInstance] accessToken] config:[[Utils sharedInstance] backendConfig]
from:[NSDate dateWithTimeInterval:-1036800 sinceDate:[NSDate date]] //one year ago
to:[NSDate date]
groupBy:AcceptStatisticsGroupByDay
resultsGroup:@[@(AcceptStatisticsOnlyMaxMin),@(AcceptStatisticsOnlyAverage),@(AcceptSt
atisticsOnlyTurnover)]
statusNesting:YES //set to NO for a simpler response
completion:^(AcceptStatistics *dict, NSError *error) { compBlock(dict, error); }];
```

```
/**
 * @brief Request of statistic data
 * @param accessToken The user session token for authentication
 * @param config Instance needed to use backend services
 * @param from starting date of the statistic range (note: response will include
automatically only dates after user's acount creation day)
 * @param to end date of the statistic range
 * @param groupBy option to group the data by type
 * @param resultsGroup array of the grouped statistic data
 * @param isNesting boolean indicating that we want a status dictionary (authorised,
rejected, reversed, etc.) per each groupBy element. NO as default
 * @param completion Block that will receive the rest of statistic data or an error
**/
- (void) queryStatistics:(NSString*)accessToken
          config:(AcceptDataServiceConfig*)config
           from:(NSDate*)from to:(NSDate*)to
        groupBy:(AcceptStatisticsGroupBy)groupBy
     resultsGroup:(NSArray *)resultsGroup
    statusNesting:(BOOL)isNesting
      completion:(void (^)(AcceptStatistics*, NSError*))completion;
```

6.1 Enumerators and classes

Please check /Documents/index.html

7. Inventory Methods

Accept backend has the option to store inventories (sets of texts, prices, ids, photos, etc) in csv, and provide the inventories through request (of course, inventories must be enabled in the backend at your disposal).

The *queryInventoryFile* method retrieves the information about inventory data available for download from the accept backend. If the inventory CSV file was uploaded to the accept backend this method returns the information about the inventory version and inventory URL it can be downloaded from.

The inventory URL can be then used in *getInventoryData* inventory to download the actual inventory data.

```
/**
 * @brief Request for the inventory file in backend
 * @param config Instance needed to use backend services
 * @param completion Block that receives the config file or an error
 * @return
 **/
- (void) queryInventoryFile:(NSString*)accessToken
            config:(AcceptDataServiceConfig*)config
        andCurrentVersion:(NSString*)version
            completion:(void (^)(AcceptInventory*, NSError*))completion;


/**
 * @brief Request to download inventory data
 * @param config:(AcceptDataServiceConfig*)config
 * @param url to download the inventory data from
 * @param completion Block that receives the data file or an error
 * @return
 **/
-(void)getInventoryData:(AcceptDataServiceConfig *)config
        theURL:(NSString *)theURL
        completion:(void (^)(BOOL, NSInteger, NSData *,
NSError*))completionAlertUI;
```

7.1 Enumerators and classes

Please check /Documents/index.html

8. Barcode Scanner and Cash Drawer

Accept SDK supports cash drawers such as the StarIO mPOP, that also print and read barcodes. They are considered as "Printers" in the SDK and not payment terminals, and due to this, they will be under the array of *supported printer vendors*. Printing with them is similar to using regular printers, though mPOP works better by sending the whole receipt as an UIImage under the *receiptImage* variable of AcceptPrinterConfig class.

Opening the drawer is pretty straightforward. As with printing and barcode reading, it is an action that can be triggered in parallel to other device ones (i.e. printing, reading and opening drawer at the same time). The accept method *openCashDrawer* will execute the opening action, of course if the drawer was already closed.

```
/**
 * @brief Opens cash drawer for supported printers (ie StarIO mPOP)
 * @param config Instance needed to use the device
 * @param completion block with the result as BOOL (YES for success) and an
error if exists
 **/
- (void) openCashDrawer:(AcceptPrinterConfig*)config
        completion:(void (^)(BOOL, NSError*))completion;
```

Barcode reader work by sampling enabling the reading using the accept method *connectBarcodeScanner*. This method receives a completion block, called when there is an error, and a block that will be triggered each time some barcode data is received.

```
/**
 * @brief Connects bascode scanner for supported printers (ie StarIO mPOP)
 * @param config Instance needed to use the device
 * @param completion block with the result as BOOL (YES for success when
connecting) and an error if exists
 * @param block that will be called each time the barcode reader gets some data,
returning it
 **/
- (void) connectBarcodeScanner:(AcceptPrinterConfig*)config
          completion:(void (^)(BOOL, NSError*))completion
        andDataReceived:(void (^)(NSData*))barcodeDataReceived;
```

Note: you only have to trigger *connectBarcodeScanner* once in the life of your accept instance, though it is better to check for unwanted lost pairing and disconnections: drawer sleeping mode may lost the pairing in some versions of iOS.

Note: StarIO barcode reader is only able to read regular barcodes, no QR codes. For a complete list of compatible formats, please search in the official page.