Developer Guide

PayPal Here SDK for iOS

Publication date:
01/12/15

PayPal

Document Revision History

Pub. date	Description of changes
12/01/15	Published

Table of Contents

Chapter 1 Overview

The PayPal Here SDK gives access to a group of PayPal transaction services. It provides an extensive set of point-of-sale functions for merchants. If you're a merchant, you can use it to develop point-of-sale apps for your own use. If you're a third-party solution provider, you can use it to develop point-of-sale software and services which you can offer to your customers.

The core features of the PayPal Here SDK are:

- Performing *card present* transactions, in which a customer presents a credit or debit card, and a merchant reads the card with a *card reader* attached to a point-of-sale terminal or smartphone. PayPal Here supports card present transactions with:
  - *Magnetic stripe ("mag stripe") cards*, currently the prevalent type of card in the United States. Mag stripe card readers are sometimes called *card swipers.*
  - *EMV cards*, a newer type of card that is being introduced in the United States, and is already widely used in other parts of the world. EMV is an acronym for "Europay Mastercard Visa," the developers of the EMV standard. EMV is also called *chip and PIN,* because an EMV card incorporates an integrated circuit (a *chip*) for improved security, and works in conjunction with a *personal identification number* (a *PIN*).
- Performing *card not present transactions*, for example, where a customer gives the merchant a credit card number by phone. Card not present transactions are also known as *manual transactions.* The card number and other data that a merchant enters to perform a card not present transaction is called *card not present data* or *keyed-in data.*
- Performing *check-in* (*tabbed*) transactions, in which a customer find a merchant's location using the PayPal Here smartphone application, *checks in* to a merchant's location (also called *opening a tab*), and pays for goods or services through a PayPal account.
- Optionally generating an *invoice* which describes the goods or services for which payment is to be made, and delivering it to the customer.

The PayPal Here SDK is implemented in these environments:

- iOS, for apps written in Objective C
- Android, for apps written in Java
- Microsoft Windows, for apps written in C#

The different implementations of the PayPal Here SDK are very similar, although not identical. If you develop point-of-sale apps in more than one environment, much of your code and your knowledge about the SDK will carry over from that environment to the others.

NOTE: The functionality described in this document is subject to change without notice.

What's in the SDK

The github repository for PayPal Here SDK for iOS contains:

- Object code and other resources for the SDK (in `./SDK/Release/PayPalHereSDK.bundle`)
- Reference (Appledoc) documentation for the SDK (in `./SDK/html/`)
- Source code for two sample apps that demonstrate use of the SDK:
  - EMVAccreditationSampleApp, which supports card present transactions with EMV cards (in `./EMVSampleApp/`)
  - SDKSampleApp, which supports card present transactions with mag stripe cards, card not present transactions with both types of cards, and check-in transactions (in `./SDKSampleApp/`)
- Source code for a sample back-end server that operates with the sample apps (in `./sample-server/`)

The SDK's place in PayPal's family of products

The PayPal Here SDK for iOS is a collection of classes, protocols, and other resources for developing point-of-sale apps that run in iOS. It communicates with the underlying PayPal APIs at the following URIs:

`https://sandbox.paypal.com/webapps/hereapi/merchant/v1`	***Sandbox environment***
`https://www.paypal.com/webapps/hereapi/merchant/v1`	***Live environment***

- The data-interchange format is JSON.
- Each request must contain an authentication token; see Authentication with the iOS SDK.

What you'll need

To install the SDK and run the sample apps, you'll need:

- An iPhone and iPad.
- iOS 6 and above
- The Xcode 5.1 integrated development environment.
- An active Apple developer account (for running the sample app on devices, in order to use the credit card reader).
- A PayPal Here card reader.

- PayPal provides a mag stripe card reader on request for any user who opens a PayPal Business or Premier account. This card reader plugs in to the iPhone's audio jack.
- A PayPal EMV card reader is available for purchase.

To develop your own apps you'll need the resources listed above, plus:

- A Log In with PayPal client ID (the same as an App ID) and secret
- Please contact us at DL-PayPal-Here-SDK@ebay.com to get the proper Scope.
- For app authentication in production, a back-end server to store your app's secret (Recommended but optional)

If you're a third-party service provider, a merchant who uses your app must have:

- Either a PayPal Business or PayPal Premier account
- An account with your own service

Note. Your app doesn't need to go through the accreditation process to perform EMV transactions. The PayPal Here SDK is already accredited with Visa and Mastercard; when your app uses the SDK it inherits the SDK's accreditation.

The basic workflow

An app must perform these setup operations to prepare to process transactions with the PayPal Here SDK:

- Initialize the SDK (each time the app starts)
- Authenticate the merchant and pass the merchant's credentials to the SDK (the first time the merchant uses the app)
- Set the active merchant, for whom the PayPal Here SDK will execute transactions
- Set the merchant's location for check-in transactions (the first time the merchant uses the app, and any time the merchant's location changes)
- Start monitoring the card reader for events (for card present transactions)

Once the setup operations are complete, an app must perform these steps to process a basic card present transaction:

- Start an invoice
- Add items to the invoice
- Take a payment using a credit card (key-in entery, Swipe, EMV card dip and more)
- Capture the customer's signature (if required for this merchant and transaction amount)
- Send a receipt

These steps are illustrated in the sample apps, and are described in more detail in Chapter 3, The transaction workflow.

Alternative workflows and additional capabilities

The SDK supports several alternatives for the steps in the basic workflow:

- A *fixed-amount transaction* instead of starting an invoice and adding items to it
- A card not present transaction instead of taking and finalizing payment with a card reader
- A check-in (tabbed) transaction using a PayPal account instead of a credit or debit card
- *Authorization and capture* (*auth/cap* for short), which enables you to authorize a transaction at one point in time and capture payment at a later point. Auth/cap is useful for businesses like restaurants, which must authorize a card payment before the customer determines the final amount by adding a gratuity.

The SDK also supports several additional capabilities:

- Issuing a refund
- Adding a referrer code to a transaction
- Adding a cashier ID to a transaction

All of the alternatives and additions to the basic workflow are described later in Chapter 3.

Important classes

Understanding the functions and roles of the PayPal Here SDK's most important classes will help you understand the detailed descriptions of how to use the SDK that come later in this guide.

PAYPALHERESDK
: The primary class for interacting with PayPal, and through the `PPHCardReaderManager` class, with hardware devices.

PPHTRANSACTIONMANAGER
: A stateful transaction processor class; takes payments and processes refunds on the current invoice (if any).

PPHCARDREADERMANAGER
: Handles interaction with mag stripe and EMV credit card readers, including audio readers, dock port readers, and Bluetooth readers.

PPHCARDREADERWATCHER
: Translates between raw card reader events and a delegate interface, Supports multiple listeners.

PPHLOCATION
: Represents a location of a merchant, along with the tabs associated with that location. Includes methods for managing locations and tabs.

PPHLocalManager
: Initiates actions on merchant locations and tabs. Listens for tab opening and closing events; initiates an action when an event is received.

PPHLocationWatcher
: Maintains a list of open tabs for a location. Includes methods for accessing and updating the list.

How this guide represents messages

This guide shows messages (as well as Objective C code) in a monospace font:

```
[tm beginRefund]
```

Placeholders (words that you must replace with appropriate values or expressions) are shown in an italic serif font:

```
[tm beginPaymentWithAmount:amount andName:name]
```

If you used this message in your app you would replace the word *amount* by a payment amount or an expression that evaluates to a payment amount, and *name* by a payment name.

Chapter 2 Before you start

Obtaining and Building the SDK

To begin using the SDK, please contact your relationship manager to get added to the GitHub repository. Confirm that you have an SSH key set up with the GitHub server. Then clone the repository and use the following command to retrieve the `Nimbus` and `AFNetworking` submodules:

```
git submodule update --init --recursive
```

Review the SDK's `Readme` file in the GitHub repository, and try building the SDK. Then review Payments with the Sample App.

Authenticating SDK operations

The PayPal Here SDK uses the OAuth 2.0 standard for authentication; that is, for confirming that an app requesting SDK services on behalf of a certain merchant is authorized to do so.

The document *Using OAuth to Authenticate Requests* describes how the OAuth 2.0 standard is used in PayPal's Mobile In-Store Payments service. Conceptually, the PayPal Here SDK uses OAuth the same way.

The authentication process has several steps:

1. Visit the PayPal developer site to create a PayPal application. This step will help you get *OAuth credentials* that you will use for authentication. When you create the application, specify the capability Log In with PayPal.

   The developer site provides an OAuth *client ID* and a *secret* for your app to use. For security, you store the secret (and the client ID too, if you prefer) on your app's back-end server, *not* in the app itself.

2. The first time a merchant uses your app, your back-end server should direct them to the PayPal authentication webpage. In addition to the endpoint's URL, the server provides a *return URL* which PayPal can use to return control to your app. The return URL should point to an endpoint on your back-end server which will return the merchant's device to your app.
3. The webpage asks the merchant to login with their PayPal credentials. Upon successful login, a terms and conditions page is shown, asking the merchant to authorize/grant your app permission to execute PayPal Here transactions on your behalf

4. Your app submits its client ID and the authorization code to the authentication endpoint through the back-end server, and obtains a long-lived *refresh token*. Upon successful authorization, PayPal would redirect the merchant to return url provided (which would typically be an endpoint in your backend-server, while also sending back an authorization code (For security, the back-end server should encrypt the token. For more information see Developing a back-end server and The sample server.)
5. Your app submits the refresh token to the authentication endpoint (again through the back-end server) to obtain a short-lived *access token*. This step must be repeated periodically when the access token expires.
6. Your app must set the current access token in the PayPalHere SDK and must repeat this process each time a new access token is obtained.

Note. The sample server and app delivered with the SDK include a test client ID and secret, along with a pre-defined test merchant. However, when you are ready to get your own client ID, you must contact your relationship manager to have your app's client ID assigned a scope that includes PayPal Here.

You must pass PayPal Here's scope identifier with each request for an access token. The scope identifier is the following URI, which is an identifier, not a link:

```
https://uri.paypal.com/services/paypalhere
```

For more information about the client ID, secret, and scope, see *Using OAuth to Authenticate Requests* and Integrate Log In with PayPal. For models of how to perform the authentication process (steps 2 through 5), see The sample server in the appendix, and the sample server's source code.

Best practices for security

Do not store your app's secret in the app on a mobile device, as it could be jail-broken or otherwise compromised. (If your app's secret is compromised, barriers are raised in your ability to provide updates to users.) Store your app's secret on the back-end server.

In principle you can use Log In with PayPal (LIPP) as your sole means of authentication. You probably have an existing account system, though, so you should authenticate the merchant in your account system, then send them to PayPal, and then link their PayPal account to their account in your system when the authentication point redirects to your return URL.

Developing a mid-tier server

You must develop a back-end server to interoperate with your app.

The essential function of the back-end server is to store your app's secret in a secure place. The back-end server may perform additional functions at your discretion.

There are two basic approaches to designing a back-end server:

- You proxy all calls to the PayPal Here SDK (including retrieving tabs, retrieving locations, making payments, etc.) through the back-end server. The access token is stored on the server.
- You only proxy the operations used to obtain access tokens through the back-end server. Your app makes other PayPal Here SDK calls directly, and the underlying services return results to it directly. The access token is stored in the app in encrypted form.

Because the back-end server's functions are simple, the sample server distributed with the SDK is a suitable (and recommended) starting point for developing your own. See 0, The sample server, for information about this server.

Chapter 3 The transaction workflow

The first part of this chapter describes the steps in setting up an app to execute transactions with the PayPal Here SDK. The second part describes the steps in executing a basic card present payment transaction.

The third section describes variations on the basic workflow (e.g., for a card not present transaction).

The fourth section describes additional capabilities that can be used in a transaction.

Setting up an app

The steps in setting up an app are:

- Initialize the SDK (each time the app starts)
- Authenticate the merchant and pass the merchant's credentials to the SDK (the first time the merchant uses the app)
- Set the merchant's location for check-in transactions (the first time the merchant uses the app, and any time the merchant's location changes)
- Start monitoring the card reader for events (for card present transactions)

Initialize the SDK

You initialize the SDK by sending a `selectEnvironmentWithType` message to `PayPalHereSDK`:

```
[PayPalHereSDK selectEnvironmentWithType:environment_type]
```

environment_type is `ePPHSDKServiceType_Sandbox` for the Sandbox environment, or `ePPHSDKServiceType_Live` for the live environment.

The SDK has a logging facility which logs various types of messages to a remote logging facility (PayPal CAL). You can choose the types of messages you want to log, and add your own decorations to the messages, by defining a class that implements the `PPHLoggingDelegate` protocol and passing an instance of it to `PayPalHereSDK.setLoggingDelegate`.

```
[PayPalHereSDK setLoggingDelegate:instance]
```

In `SDKSampleApp`, `STAppDelegate.m` implements the protocol methods and sets the logging delegate to `self`.

Authenticate the merchant

Authentication is described in Authenticating SDK operations. Most of the steps involve calls to OAuth (see *Using Oath to Authenticate Requests*) or to the sample back-end server.

Set the active merchant

Once the app has authenticated the server, it calls `PayPalHereSDK.setActiveMerchant` to set the merchant for which transactions will be executed.

```
[PayPalHereSDK setActiveMerchant:merchant withMerchantId:merchantId completionHander:handler]
```

- ***merchant*** is an instance of the PPHMerchantInfo represeting a merchant object
- ***merchantId*** is an *id* for the merchant. It is defined by agreement between the back-end server and the app (not by the SDK), and must be unique among the merchants that use the back-end server and the app.
- ***handler*** is an *id for* a completion handler to be called when merchant setup is completed.

Set the merchant's location

In the `SettingsViewController.m` sample file, see the code that uses the `PPHLocation` class, which includes functionality for getting a list of previous merchant locations.

> **NOTE:** If your app performs check-in transactions, location services must be enabled for it at all times. Your app should prompt the user to enable location services for it if necessary. Even if there is a failed attempt to check in the merchant (i.e. set the merchant's location), your app can take payments (except check-in payments).

Start monitoring the card reader

Once the merchant is ready to take a payment, the application should invoke the below API:

`[[PayPalHereSDK sharedCardReaderManager] beginMonitoring]; (SettingsViewController.m).`

This would tell the SDK to start monitoring the connected external accessory, such as the card swipe reader, EMV terminal etc for any swipes or card insertions respectively.

Basic workflow for a transaction

These sections describe the basic steps in a card payment transaction. Later sections describe possible variations on this workflow, and other workflows.

Start an itemized transaction (an invoice)

```
// Creating a simple invoice and starting a transaction.
PPHTransactionManager *tm = [PayPalHereSDK sharedTransactionManager];
    PPHAmount *total = [PPHAmount amountWithString:self.amount];
    [tm beginPaymentWithAmount:total andName:@"simplePayment"];
```

(STCardSwipeViewController.tm)

Add items to the invoice

```
PPHInvoice *invoice = [[PPHInvoice alloc] initWithCurrency:@"USD"];

    NSString *taxRate = [[NSUserDefaults standardUserDefaults]
objectForKey:@"taxRate"];
    NSDecimalNumber *taxRateNumber;
    if (taxRate) {
        taxRateNumber = [NSDecimalNumber
decimalNumberWithString:taxRate];
    } else {
        taxRateNumber = [NSDecimalNumber
decimalNumberWithString:@".10"];
    }

    for (NSString *item in self.shoppingCart) {
        [invoice addItemWithId:item detailId:nil name:item
quantity:self.shoppingCart[item] unitPrice:self.store[item]
taxRate:taxRateNumber taxRateName:@"taxRate"];
    }
```

(TransactionViewController.m)

```
// Begin the purchase and forward to payment method
    PPHTransactionManager *tm = [PayPalHereSDK
sharedTransactionManager];
    tm.ignoreHardwareReaders = NO;
    [tm beginPayment];
    tm.currentInvoice = invoice;
```

(SimpleFSPaymentDelegate.m)

Take a payment using a credit card reader

In order to start processing magstripe based payments, the class/view controller that would invoke the processPayment API, it is required the same to also implement the "PPHTransactionManagerDelegate". This protocol would receive events (via the "onPaymentEvent" method)from the SDK, describing the steps being performed during the payment lifecycle.

In order to implement the above "onPaymentEvent" method and receive events off it, init an instance of the "PPHTransactionWatcher" class.

```
[[PPHTransactionWatcher alloc] initWithDelegate:self];
```

Once we have an active invoice in the transaction manager and have invoked the beginMonitoring API, upon a card swipe on the reader, the onPaymentEvent method would be invoked with a parameter of type PPHTransactionManagerEvent.

```
if (event.eventType == ePPHTransactionType_CardDataReceived)  {

            //Now ask to authorize (and take) payment all in one shot.
            [[PayPalHereSDK sharedTransactionManager]
processPaymentWithPaymentType:ePPHPaymentMethodSwipe

withTransactionController:self

completionHandler:^(PPHTransactionResponse *response) {

self.transactionResponse = response;
                                                                    if
(response.error) {

[self showPaymentCompeleteView];
                                                                    }
else {

// Is a signature required for this payment?  If so

// then let's collect a signature and provide it to the SDK.

if (response.isSignatureRequiredToFinalize) {

[self collectSignatureAndFinalizePurchaseWithRecord];

 } else {
```

```
// All done.  Tell the user the good news.

//Let's exit the payment screen once they hit OK

_doneWithPayScreen = YES;

[self showPaymentCompeleteView];

 }
}
```

(PaymentMethodViewController.m)

Finalize a payment with a customer signature

```
[[PayPalHereSDK sharedTransactionManager]
provideSignature:self.signature.printableImage

forTransaction:_capturedPaymentResponse.record

completionHandler:^(PPHError *error) {
                        [self
showPaymentCompeleteView:_capturedPaymentResponse];
                    }];
```

(SignatureViewController.m)

Send a receipt

```
    PPHTransactionManager *tm = [PayPalHereSDK
sharedTransactionManager];
    PPHReceiptDestination * destination = [[PPHReceiptDestination
alloc] init];
    destination.destinationAddress = infoString;
    destination.isEmail = _isEmail;
    [tm sendReceipt: _transactionRecord toRecipient: destination
completionHandler:^(PPHError *error) {
        if (error == nil) {
            [self showAlertWithTitle:@"Receipt Sent"
andMessage:@"Please wait for a few minutes to receive the receipt on
your device."];
        } else {
            [self showAlertWithTitle:@"Error while sending receipt."
```

```
andMessage:error.description];
        }
    }];


(ReceiptInfoViewController.m)
```

A card not present transaction

```
PPHCardNotPresentData *manualCardData = [[PPHCardNotPresentData alloc]
init];
    manualCardData.cardNumber = cardNumStr;
    manualCardData.cvv2 = cvvStr;
    manualCardData.expirationDate = [[NSCalendar currentCalendar]
dateFromComponents:comps];

//Now, make a payment with card data
    PPHTransactionManager *tm = [PayPalHereSDK
sharedTransactionManager];
    PPHAmount *total = [PPHAmount amountWithString:self.amount];
    [tm beginPaymentWithAmount:total andName:@"simplePayment"];
    tm.manualEntryOrScannedCardData = manualCardData;



[tm processPaymentWithPaymentType:ePPHPaymentMethodKey
                withTransactionController:nil
                        completionHandler:^(PPHTransactionResponse
*record) {
    if (record.anyError) {
// Error during transaction.
    } else {
// Payment successful.
    }
];
```

A check-in transaction

For check-in transactions, Please refer to chapter 5 (SDK Features for check-in transactions) for the initial merchant location setup.

Once the merchant has successfully checked in, PayPal would generate and return a locationId. The application should store this value locally, preferably in the app delegate, as it would be required while searching for nearby customers who are checked-in to PayPal on the merchant's store location.

In order to retrieve a list of checked-in customers, the app would need to create an instance of the PPHLocationWatcher and implement the methods in the `PPHLocationWatcherDelegate`.

```
self.checkinLocationId = appDelegate.merchantLocation.locationId;
        self.locationWatcher = [[PayPalHereSDK sharedLocalManager]
watcherForLocationId:self.checkinLocationId withDelegate:self];
```

The sample app displays the list of checked in customers via a table view:

```
#pragma mark PPHLocationWatcherDelegate
-(void)locationWatcher:(PPHLocationWatcher *)watcher
didCompleteUpdate:(NSArray *)openTabs wasModified:(BOOL)wasModified
{
    NSLog(@"Got the response didCompleteUpdate from Location Watcher
with list of checked-in clients. No. of clients: %ld", (unsigned
long)[openTabs count]);
    self.checkedInClients = [[NSMutableArray alloc]
initWithArray:openTabs];
    [self.tableView reloadData];
}

-(void)locationWatcher: (PPHLocationWatcher*) watcher didDetectNewTab:
(PPHLocationCheckin*) checkin
{
    NSLog(@"Got the new checked in client after did the update. Need
to update the rows");
}

-(void)locationWatcher: (PPHLocationWatcher*) watcher
didDetectRemovedTab: (PPHLocationCheckin*) checkin
{
    NSLog(@"One of the checked in client checked out. Need to update
the rows");
}

-(void)locationWatcher: (PPHLocationWatcher*) watcher didReceiveError:
(PPHError*) error
{
    NSLog(@"Oops got the error while looking for checked in
clients..");
}
```

When a customer (table cell) is selected:

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    PPHLocationCheckin *client = [self.checkedInClients
objectAtIndex:indexPath.row];
    if (nil != client){
        NSLog(@"Calling takePaymentUsingCheckinClient with the
```

```
checkedin client: %@",client.customerName);
        [self takePaymentUsingCheckinClient:client];
    } else {
        NSLog(@"Oops! Selected row has no checkeding client..");
    }
}



// Taking payment

-
(void)takePaymentUsingCheckinClient:(PPHLocationCheckin*)checkinMember
{

    PPHTransactionManager *tm = [PayPalHereSDK
sharedTransactionManager];
    [tm setCheckedInClient:checkinMember];
    [tm processPaymentWithPaymentType:ePPHPaymentMethodPaypal
            withTransactionController:self
                    completionHandler:^(PPHTransactionResponse  *record)
{
                        [self.processingTransactionSpinny
stopAnimating];
                        self.processingTransactionSpinny.hidden=YES;
                        if (record.error) {
                            NSString *message = [NSString
stringWithFormat:@"Payment using checkin Failed with an error: %@",
record.error.apiMessage];
                            [self showAlertWithTitle:@"Payment Failed"
andMessage:message];
                        } else {
                            self.transactionResponse = record;

                        }

                    }];

}
```

Authorization and capture

Authorization and capture (*auth/cap* for short) lets you authorize a transaction and then capture the payment at a later time. This enables you to authorize a transaction before its exact amount is known.

In one common use case a merchant accepts a credit card from a customer, authorizes a transaction, and presents a receipt to the customer, who adds a gratuity. The merchant can then compute the total amount and capture payment.

Setting up a payment authorization

All auth/cap requests use an access token for authentication, the same as the rest of the PayPal Here SDK. For an outline of PayPal Here authentication procedures and references to more detailed information, see Authenticating SDK operations.

To set up a payment authorization for a transaction:

1. Initialize the SDK, start a transaction, set an amount, and define an invoice in the usual way.
2. Instead of taking payment in the usual way, call the transaction manager's `authorizePaymentWithPaymentType` method:

```
[tm authorizePaymentWithPaymentType:ePPHPaymentMethodKey
                  withCompletionHandler:^(PPHTransactionResponse *response) {
                      if (!response.error) { self.auth = response.record }
                  }
];
```

   `ePPHPaymentMethodKey` determine the type of payment. The second parameter specifies a standard completion handler.

3. If the completion handler receives a "no error" response, the response's `record` property contains a pointer to a `PPHTransactionRecord`. Save the pointer for later use.

Working with authorized payments

Once you have authorized a payment, you can:

- Capture the authorized payment
- Void the authorization (before payment is captured)
- Refund part or all of the payment (after payment is captured)

Capturing payment

To capture payment for an authorized transaction, call `capturePaymentForAuthorization`:

```
[tm capturePaymentForAuthorization:transactionRecord
        withCompletionHandler:^(PPHTransactionResponse *response) {
        }
];
```

transactionRecord is the transaction record returned from the authorization call.

The second parameter is a standard completion handler.

The amount captured is determined by the invoice (the PPHInvoice object) associated with the ***transactionRecord***. The amount may exceed the original authorized amount by a percentage to allow for adding a typical gratuity. The percentage is determined by the type and characteristics of the merchant, and cannot be changed through the SDK.

Voiding an authorization

To void an authorization before the payment has been captured, call the transaction manager's `voidAuthorization` method:

```
[tm voidAuthorization:transactionRecord
        withCompletionHandler:^(PPHTransactionResponse *response) {
        }
];
```

transactionRecord is the transaction record returned from the authorization call.

The second parameter is a standard completion handler.

Refunding a captured payment

See the Refunding a payment transaction section below.

EMV related transactions:

In order to perform an EMV transaction, please make sure the EMV terminal is paired and connected. Also, make sure the terminal is installed with the latest version of the EMV software provided by PayPal. Once ready, the below sample shows an example on taking a payment via the EMV SDK:

```
PPHAmount *amount = [PPHAmount amountWithDecimal:decimalAmount];
            [tm beginPaymentWithAmount:amount
andName:@"accreditationTestTransactionItem"];


[tm processPaymentUsingSDKUI_WithPaymentType:ePPHPaymentMethodChipCard
                                withTransactionController:nil
                                       withViewController:self

completionHandler:^(PPHTransactionResponse *record) {
                if (record && !record.error) {
// Payment successful.
```

```
                } else {
// There was an error processing the payment.
                }

                }];
```

(EMVTransactionViewController.m)

The above API would perform the EMV transaction and also, provide the UI for sending the receipt to a customer.

Refunds on a completed transaction could be performed via:

```
[[PayPalHereSDK sharedTransactionManager]
beginRefundUsingSDKUI_WithPaymentType:ePPHPaymentMethodChipCard
withViewController:self record:self.record amount:self.amount
completionHandler:^(PPHTransactionResponse *response) {

        if (response.error) {
// error during refund.
          } else {
// refund successful.
          }

    }];
```

(EMVSalesHistoryViewController.m)

Additional capabilities

This section describes several additional capabilities of the SDK which you can use with transactions.

Refunding a payment

Mag stripe transactions (SDKSampleApp)

To refund a payment, call the `PPHTransactionManager` class's `beginRefund` method.

```
[tm beginRefund:transactionID forAmount:amount completionHandler:handler];
```

transactionID is the transaction ID of the transaction to be refunded.

amount is a `PPHAmount` that specifies the amount to be refunded. To refund the entire amount of the transaction, use `nil`.

handler is a completion handler.

EMV transactions (EMVAccreditationSampleApp)

To refund a payment, call the `PPHTransactionManager` class's `beginRefundUsingSDKUI_WithPaymentType` method.

```
[tm beginRefundUsingSDKUI_WithPaymentType:ePPHPaymentMethodChipCard
        withViewController:viewController record:transactionRecord amount:amount
        completionHandler:handler];
```

`ePPHPaymentMethodChipCard` indicates the type of payment being refunded (a payment with an EMV card).

viewController is a class that implements…

transactionRecord is the transaction record returned by the transaction to be refunded.

amount is a `PPHAmount` that specifies the amount to be refunded. To refund the entire amount of the transaction, use `nil`.

handler is a completion handler.

The method returns a new transaction record that represents the refund.

Authorized captured payments

The procedure for refunding an authorized, captured payment is described under *Refunding a captured payment* in the section *Authorization and capture*.

Adding a referrer code (BN code or attribution code)

A developer partner's integration with PayPal can be identified with a *referrer code* (also known as a *build notation code*, *BN code*, or *attribution code*). If your app sets a referrer code when it initializes the PayPal Here SDK, the SDK will include the referrer code in each call that it makes to a PayPal API.

A referrer code can help you capture information about your clients' use of your software. This information is essential if your organization charges use-based fees, and it can also help you analyze how your software is being used.

You can obtain a referrer code from your account manager. You can set it during SDK setup by calling the `setReferrerCode` method of the `PayPalHereSDK` class.

```
[PayPalHereSDK setReferrerCode withCode:referrerCode]
```

referrerCode points to your referrer code, which is ordinarily an NSString.

Adding a cashier ID

You can identify a merchant "sub-user" in an invoice by specifying a cashier ID. The sub-user is typically the person immediately responsible for the transaction, e.g. a cashier who takes a payment.

To include a cashier ID in an invoice, store it as an `NSString` in the `PPHInvoice` object's `cashierId` property:

```
mInvoice.cashierId = @"cashierID";
```

In this example, mInvoice represents a PPHInvoice object, and *cashierID* is a cashier ID.

Making calls directly to the Mobile In-Store Payments API

Apps that use PayPal software only to read credit cards, and not to perform transactions within the PayPal Here framework, must call the Mobile In-Store Payments API directly rather than use the PayPal Here SDK.

The Mobile In-Store Payments API is a RESTful interface with the following characteristics:

- The data-interchange format is JSON.
- Each request must contain an authentication token; see Authenticating SDK operations.
- The base URI is `https://www.paypal.com/webapps/hereapi/merchant/v1` .

For more information about the Mobile In-Store Payments API, see Customizing the SDK with a transaction controller and the *Mobile In-Store Payments API Developer Guide*.

Chapter 4 Using credit card data

As described in the Overview, your app can take credit card payments with card data from a card reader or with card data that is manually keyed in.

Fees for keyed-in data are higher than fees for swipes; see the PayPal Here FAQs for details.

To take credit cards using the SDK, the merchant must be approved for PayPal Here. After authenticating a merchant, but before taking a payment, use the `status` property of the `PPHAccessAccount` class to determine whether the logged-in merchant is approved for PayPal Here.

Handling keyed-in card data

If a merchant manually enters data from a credit card, instead of swiping it, the data is *keyed-in data*, also called *card not present data*.

When an app accepts keyed-in data, the app must fill in the appropriate fields of a `PPHCardNotPresentData` object and provide that object to a `PPHTransactionManager` object to capture the payment. In this case, the required properties of the `PPHCardNotPresentData` object are:

- `cardNumber`
- `expirationDate`
- `cvv2`

Specifying a valid `postalCode` greatly increases the likelihood that keyed-in data will be accepted. To support international cards, the postal code must allow alphanumeric values.

Validating the card data

Your app should validate the card number and date before providing the `PPHCardNotPresentData` object to the SDK. Note that the SDK handles the date format through the `NSDate` object, which requires a four-digit year. Your app should validate that the expiration month/year combination is the current month or a future month.

The card number should be between 13 and 19 digits long. Use the standard Luhn algorithm (also called the *mod 10 algorithm* or the *modulus 10 algorithm*) to sanity-check the card number.

Presenting the card number and date to the user

Your app's user interface should automatically space the card number in groups of four digits as the user enters it. (Note that American Express cards, whose *bank identification numbers* (BINs) start with 34 or 37, have 15 digits. For these numbers, space the number in groups of 4, 6, and 5 digits.

Present the expiration date to the user as it is printed on a typical card: a two-digit month followed by a two-digit year. Preferably, let the user enter the month and year from a pair of drop-down lists with a slash ("/") between them.

Accepting a Signature

After your app completes an invoice and receives a card-read notification, but before it pays the invoice, it can capture a signature image. (For some merchants and transaction-amounts, the PayPal Here SDK requires a signature before payment).

Use the `PPHTransactionManager` class's `finalizePaymentForTransaction` method, which finalizes the payment.

Detailed Communication with a Credit Card Reader

This section is an overview of how an app uses the SDK for detailed communication with a credit card reader. "Detailed communication" means that the app controls individual card reader operations, rather than simply asking the SDK to read a card.

> **NOTE:** Your app can use the `PPHTransactionManager` class to take credit card payments, **without** this type of detailed communication.

Starting Communication with the Card Reader

The `PPHCardReaderManager` class handles all interaction with all types of credit card readers, including audio readers, dock port readers, and Bluetooth readers. It handles readers for both mag stripe and EMV cards.

Monitoring for card reader connections

To communicate with a card reader, your app must first start *monitoring* for connection or removal of card readers. The `PPHCardReaderManager` class's `beginMonitoring` method does this. The PayPalHereSDK class's sharedCardReaderManager property points to a default card reader manager which you can use for this purpose.

```
[[PayPalHereSDK sharedCardReaderManager] beginMonitoring];
```

`beginMonitoring` accepts a parameter which specifies the types of readers to monitor.

```
[[PayPalHereSDK sharedCardReaderManager] beginMonitoring:readerTypes];
```

readerTypes is an `id` that points to a `PPHReaderTypeMask` value.

beginMonitoring looks for events from any devices that is connected such as an inserted the phone or a Bluetooth connected terminal.

> **NOTE:** If you are using a PayPal-supported, custom accessory swiper (such as a Magtek custom branded swiper), set it up using the `PPHCardReaderBasicInformation` class before `PPHCardReaderManager`'s `beginMonitoring` method.

Handling card reader notifications

After your app begins monitoring for card readers, the SDK will fire notification center events as it discovers readers.

Rather than monitor the notification center directly, you should make use of the protocol that translates untyped notification center calls into typed delegate calls. Simply store an instance of `PPHCardReaderWatcher` in your view controller class and make the class implement the `PPHCardReaderDelegate` protocol:

```
self.readerWatcher =
    [[alloc] initWithDelegate: self];
```

Your app will be notified when the SDK starts monitoring for card readers, when a card reader is connected or removed, when card reader metadata is received, and when a card swipe (or EMV equivalent) is attempted, completed, and failed. See the Appledoc description of `PPHCardReaderWatcher` for details.

Because audio jack readers have batteries in them, be careful about leaving a `PPHTransactionManager` object open for too long. For information see the `PPHTransactionManager` class.

Activating a card reader

When your app is notified that a card reader of interest to you has been connected, it should connect or activate the reader. The `PPHCardReaderManager` class's `activateReader` method does this. In the case of an audio reader, the battery may be activated; in other cases, an activity such as connecting to a Bluetooth accessory or feature port accessory is completed:

```
[[PayPalHereSDK sharedCardReaderManager] activateReader:readerOrNil];
```

readerOrNil points to a `PPHCardReaderBasicInformation` object which identifies the reader to activate. A `nil` represents the default reader or the only reader.

> **NOTE:** Your app should have permission to access to the user's GPS coordinates from Apple's location services before you activate a reader because activation requires access to the device's GPS coordinates.

The SDK uses high volume audio tones to communicate with audio readers. Therefore, your app should confirm that a user is not likely to have head phones plugged into the audio jack when your app starts monitoring for card readers.

Chapter 5 SDK features for check-in transactions

Check-in transactions are transactions which are paid through a PayPal account rather than a credit or debit card. They are unique to PayPal, and offer several features that traditional card transactions do not.

A customer begins a check-in transaction by displaying a list of near-by merchant locations in the PayPal Wallet app on their smartphone. The customer selects a merchant location, and thereby *checks in* to that location. Checking in creates a *tab* which represents the state of the transaction.

When the customer is ready to pay for goods and services they have purchased, a merchant employee (a cashier) identifies them from a list of checked-in customers, that is, customers who have open tabs. PayPal Here causes the customer's PayPal Wallet app to display a request to pay the tab. If the customer agrees to the request, PayPal Here pays the merchant through the customer's PayPal account.

Because the workflow of a check-in transaction is different from that of a card transaction, it causes a merchant app to use different classes and methods in the PayPal Here SDK.

Check-in payment workflow

PayPal Here uses location data extensively to process check-in transactions, and in fact cannot process them at all unless location services are enabled for the customer's PayPal Here app. For example, PayPal Here needs to know the customer's location to display the list of near-by merchants who offer check-in transactions. Thus, for check-in transactions the PayPal Here SDK's location management features are important.

A merchant app can create a location for a merchant, get current locations for the merchant, and modify location properties such as latitude and longitude and whether a location currently is open.

For more information about how PayPal Here uses location data, see *Mobile In-Store Locations API: Managing Merchant Locations.* (Although *Managing Merchant Locations* describes an API associated with the Mobile In-Store Payments service, the PayPal Here SDK uses location data in similar ways.)

An app that processes check-in payments should use a workflow similar to this one:

1. During merchant setup, create a location to represent the merchant's place of business (or each of them). The coordinates of a location may be derived from the host device's GPS coordinates, or may be keyed in.

2. In operation, periodically retrieve customers who have checked in (who have opened tabs) at the location.
3. Allow the merchant to select a customer who wants to pay their tab.
4. Create an invoice for the customer, select items for the invoice, and confirm the invoice amount.
5. Accept a PayPal payment for the amount of the invoice.

For details about how the PayPal Here SDK calls are used to manage location data, see the Javadoc pages for the classes `PPHLocalManager`, `PPHLocation`, and other classes whose names start with `PPHLocation`.

Tab data

The PayPal Here SDK offers a rich set of classes and methods for managing data about tabs. For example, you can create, update, and delete locations, set the location where an instance of your app is running, and fetch a list of tabs that are open at that location. You can also watch the location for tabs being opened and closed.

You use fetch a `PPHLocationCheckin` object to get information about a tab, such as its status, its amount, and its customer's ID, name, and photo URL.

You also can watch for newly opened and deleted tabs at a location with the `PPHLocalManager` class's `watcherForLocationId:withDelegate` method.

The following code gets a list of locations and modifies the first location. Then the code creates and saves a location watcher, which will monitor the location for new tabs.

```
[
    [PayPalHereSDK sharedLocalManager]
    beginGetLocations:^(PPHError *error, NSArray *locations) {
        PPHLocation *loc = [locations objectAtIndex:0];
        loc.contactInfo.lineOne = @"1 International Place";
        loc.contactInfo.city = @"Boston";
        loc.contactInfo.state = "MA";
        loc.contactInfo.countryCode = @"US";
        loc.tabExtensionType = ePPHTabExtensionTypeNone;
        [loc save:^(PPHError *error) {
            self.locationWatcher = [
                [PayPalHereSDK sharedLocalManager]
                        watcherForLocationId:loc.locationId withDelegate:self
            ];
            [self.locationWatcher update];
        } ];
    }
];
```

For details about the classes for managing tabs, see the reference documentation for the `PPHLocation` class and other classes whose names begin with `PPHLocation`.

Chapter 6 Creating an invoice

An *invoice* is represented by the `PPHInvoice` class. You create an invoice by creating a new instance of `PPHInvoice`. You should set a currency type and add one or more items, as in the example below, and set the tax or any other required information:

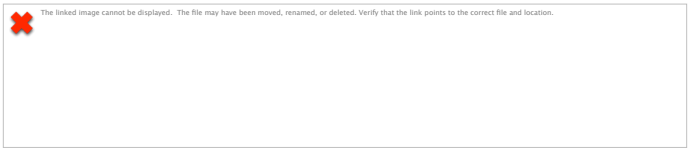
```
PPHInvoice *invoice = [ [PPHInvoice alloc] initWithCurrency:self.currencyField.text ];
[invoice addItemWithId: @"Item"
        name: @"Purchase"
        quantity: [NSDecimalNumber one]
        unitPrice: [PPHAmount amountWithString:self.amountField.text
                            inCurrency:self.currencyField.text].amount
        taxRate: nil
        taxRateName: nil
];
```

You can also create an invoice by calling the `PPHTransactionManager` class's `beginPayment` method. This call creates an invoice and makes the SDK begin watching the card reader for card swipes at the same time.

For information about the possible contents of an invoice (item quantity, unit price, etc), see the description of the `invoice` object in the REST API Reference.

Please note the following requirements for setting item prices and tax rates:

- Store tax rates as string values, not floating point values.
- Use the following rule for tax rounding: round to the nearest penny (to two digits after the decimal point) at the item level. In the following example, the correct amount for tax is 0.61:



Add details about each invoice item on the receipt, if possible.

Chapter 7 Customizing the SDK with a transaction controller

The transaction controller can be used to customize the SDK. Specifically, you can use the `PPHTransactionControllerDelegate` to use the Mobile In-Store Payments API directly.

For example, if your app utilizes a custom credit card reader, you could implement the `PPHTransactionControllerDelegate` to call the Mobile In-Store Payments API directly (instead of through the SDK).

The `PPHTransactionControllerDelegate` has two methods that intercept authorize events, as follows. To call the Mobile In-Store Payments API directly, your app would override these two methods:

- `onPreAuthorizeForInvoice`. Invoked by the SDK right before the SDK calls the Mobile In-Store Payments API, for taking a payment (authorizing a payment). The PPHInvoice object and the request payload string (invoice ID and card data, in `preAuthJSON`) are passed to the `onPreAuthorize` method.
- `onPostAuthorize`. Invoked by the SDK after a payment call, the `onPostAuthorize` method passes the following value:
    - A `didFail` boolean to indicate whether the attempted authorization was successful

If you want the SDK to collect the card data, and make payment API calls, your app could either:

- *Refrain from implementin*g the `PPHTransactionControllerDelegate`, or
- Implement the `PPHTransactionControllerDelegate` and in the `onPreAuthorizeForInvoice` method, return the `PPHTransactionControlActionType.ePPHTransactionType_Continue` enum, to tell the SDK that it must make the payment-related API call

If you want your app to fill in the card data and make its own payment API call, your app could use the `onPreAuthorizeForInvoice` string, containing the missing card data, and make the payment API call.

For an example of how to implement a transaction controller for pre- and post-authorize events, see the `CCCustomInputViewController`.m.

Below is an example of using the PPHTransactionControllerDelegate and letting the SDK handle the payment process.

```
#pragma mark PPHTransactionControllerDelegate
-(PPHTransactionControlActionType)onPreAuthorizeForInvoice:(PPHInvoice
*)inv withPreAuthJSON:(NSString*) preAuthJSON
{
    NSLog(@"TransactionViewController: onPreAuthorizeForInvoice
called");
    return ePPHTransactionType_Continue;
}

-(void)onPostAuthorize:(BOOL)didFail
{
    NSLog(@"TransactionViewController: onPostAuthorize called.
'authorize' %@ fail", didFail ? @"DID" : @"DID NOT");
}
```

```
PPHTransactionManager *tm = [PayPalHereSDK sharedTransactionManager];

    [tm processPaymentWithPaymentType: ePPHPaymentMethodSwipe
            withTransactionController:self
                    completionHandler:^(PPHTransactionResponse  *record)
{
                        _doneWithPayScreen = YES;   //Let's exit the
payment screen once they hit OK
                        self.transactionResponse = record;
                        [self showPaymentCompeleteView];
                        tm.ignoreHardwareReaders = NO;    //Back to
the default running state.
                    }];
```

(PaymentMethodViewController.m)

Here is an example of the app taking over the payment processing:

```
-(PPHTransactionControlActionType)onPreAuthorizeForInvoice:(PPHInvoice
*)inv withPreAuthJSON:(NSMutableDictionary*) preAuthJSON {

    UIActivityIndicatorView *spinny = [[UIActivityIndicatorView alloc]
initWithActivityIndicatorStyle:UIActivityIndicatorViewStyleGray];
    [spinny setFrame:CGRectMake(0, 0, 100, 100)];
    [spinny startAnimating];
    UIBarButtonItem *loading = [[UIBarButtonItem alloc]
```

```
initWithCustomView:spinny];
    self.navigationItem.rightBarButtonItem = loading;

    STAppDelegate *appDelegate = (STAppDelegate *)[[UIApplication
sharedApplication] delegate];

    NSURL *url = [NSURL URLWithString:[NSString
stringWithFormat:@"%@%@", appDelegate.serviceURL,
@"/webapps/hereapi/merchant/v1/pay"]];
    NSMutableURLRequest *request = [NSMutableURLRequest
requestWithURL:url];
    NSData *jsonData =  [NSJSONSerialization
dataWithJSONObject:preAuthJSON
                                                        options:0
                                                          error:nil];
    [request setHTTPMethod:@"POST"];
    [request setHTTPBody:jsonData];
    [request setValue:[NSString stringWithFormat:@"Bearer %@",
[PayPalHereSDK activeMerchant].payPalAccount.access_token]
forHTTPHeaderField:@"Authorization"];
    [request setValue:@"application/json; charset=UTF-8"
forHTTPHeaderField:@"Content-Type"];



    [NSURLConnection sendAsynchronousRequest:request
queue:[NSOperationQueue mainQueue] completionHandler:^(NSURLResponse
*response, NSData *data, NSError *error) {
        PPHTransactionResponse *transactionResponse =
[[PPHTransactionResponse alloc] init];
        if (error) {
            transactionResponse.error = [PPHError
pphErrorWithNSError:error];
        } else {
            NSDictionary *dict = [NSJSONSerialization
JSONObjectWithData:data options:0 error:nil];
            transactionResponse.record = [[PPHTransactionRecord alloc]
initWithTransactionId:dict[@"transactionNumber"]
andWithPayPalInvoiceId:dict[@"invoiceId"]];
        }
        PaymentCompleteViewController *vc =
[[PaymentCompleteViewController alloc]
initWithNibName:@"PaymentCompleteViewController" bundle:nil
forResponse:transactionResponse];
        [self.navigationController pushViewController:vc animated:YES];
    }];

    return ePPHTransactionType_Handled;
}
```

(CCCustomInputViewController.m)

Chapter 8 Handling errors

Many PayPal Here SDK calls perform asynchronous operations. These calls typically expect a parameter that contains a *completion handler*, which is an instance method or code block to be called when asynchronous operation completes or fails. A completion handler expects one parameter that points to a PPHError object.

This example shows how a completion handler might be defined in a call to the `PPHLocation` class's `save` method:

```
PPHLocation *loc = [locations objectAtIndex:0];
[loc save:^(PPHError *error) {
    self.locationWatcher = [
        [PayPalHereSDK sharedLocalManager]
                watcherForLocationId:loc.locationId withDelegate:self
    ];
    [self.locationWatcher update];
} ];
```

If the operation completed, the completion manager's `error` parameter is nil. If the operation failed, it points to a `PPHError` object constructed by the SDK.

The following sections describe the properties of `PPHError`, which provide potentially useful information about the error.

For information about specific SDK calls' errors and recommended responses to them, see PayPal Here SDK's reference documentation for `PPHError` and the comments in `PPHError.h`.

The `domain` property

`domain` indicates where an error originated (e.g. in one of the PayPal Here SDK's underlying services or in the app). The property can have one of several values:

- `PPHServer` indicates an error that originated in one of the underlying services.
- `PPHLocal` indicates an error that originated locally, that is, in the SDK itself or in the app.
- `PPHHTTP` indicates an HTTP error, that is, a problem somewhere in the communication path between the app and one of the underlying services.
- `PPHInvoice` indicates that the app failed to save an invoice, or an invoice operation requested by the app was canceled.
- Any other value indicates an iOS error. The SDK passes such errors through to the application.

The `code` property

`code` indicates determine what specific error occurred. Its value is a number (an error code) which identifies the error.

For a local error, the code is defined in the SDK and you can check against it.

For example, consider the following subset of error codes (defined in PPHTransactionManager.h) which `PPHTransactionManager` can return:

```
#define kPPHLocalErrorBadConfigurationNoCheckedInClient -2000
#define kPPHLocalErrorBadConfigurationNoCardData -2001
#define kPPHLocalErrorBadConfigurationNoManualCardData -2002
#define kPPHLocalErrorBadConfigurationNoMerchant -2003
#define kPPHLocalErrorBadConfigurationNoRecord -2004
#define kPPHLocalErrorBadConfigurationInvalidState -2005
#define kPPHLocalErrorBadConfigurationMerchantAccountNotActivatedForPayments -2006
#define kPPHLocalErrorBadConfigurationInvalidParameters -2007
#define kPPHLocalErrorBadConfigurationNoCurrentInvoice -2008
#define kPPHLocalErrorBadConfigurationInvoiceAlreadyPaid -2009
#define kPPHLocalErrorBadConfigurationNoInvoiceInTransactionRecord - 2011
```

These error codes often indicate a condition which the application could address by correcting their usage of the SDK.

Note that error code -2000 means that your app asked the SDK to take a check-in payment, but forgot to tell the Transaction Manager which checked-in client to charge.

The `apiMessage` property

For a service error, `apiMessage` often contains a description of the error addressed to the user. Such a message is generally suitable to be displayed in the UI.

Local errors, originating in the SDK, typically don't have an `apiMessage` value.

The `devMessage` property

`devMessage` often contains a description of the error addressed to the application developer. Such a description is generally *not* suitable to be displayed in the UI (i.e. to an end user). You can get access to a `devMessage` value by recording it in a log, or by processing either of these messages:

```
NSString *devMessage = [error.userInfo objectForKey:@"DevMessage"];
NSString *devMessage = error.devMessage;
```

How to process errors

Your error processing design should be guided by the questions:

- What does this error mean to the user?
- Can the app resolve it, or at least make it easier for the user to resolve?
- What can the user do to resolve it?

Depending on the answers to these questions, it may be appropriate for the app to do one or more of these things:

- Treat the error as an event that is exceptional but not "wrong" (not really an error) and handle it automatically
- Report the error in terms that meaningful to the user
- Offer the user a reasonable set of options for responding to the error
- Log the error, tell the user that something has gone wrong, and partially or completely abandon the operation that the app was trying to perform
- Do anything else that makes sense for the app and its intended user

The more error conditions the app handles, and the more completely it handles them, the better the user experience it will provide.

A `PPHServer` error often represents a condition that is not an error from the app's point of view. Such a condition should be handled by the app — it should not be logged or displayed to the user as the `PPHServer` object describes it.

For example, an "error" that indicates "card declined" or "charge exceeds merchant limit" should make the app report the condition to the user and offer appropriate response options.

A `PPHLocal` error or iOS error (with a "none of the above" value in `domain`) is usually caused by improper use or configuration of the SDK. Make the app log the error's details. When you review the log, think about whether you can change the app to prevent the error from occurring.

A `PPHHTTP` error may mean that the app does not have access to the Internet (prompt the user to enable WiFi or 3G), that one of the underlying services isn't available (ask the user to be patient), or that the service has denied access to the app (try again later; contact PayPal user support if it still doesn't work).

`PPHInvoice` errors fall into two categories. An error that follows a "cancel invoice" operation is not really an error — it's just a confirmation that the invoice was canceled successfully. An error that follows an attempt to make a payment or save an invoice is similar in nature to a `PPHLocal` error, and should be handled similarly.

Appendix A. The sample apps

The PayPal Here for iOS SDK is delivered with two sample apps:

- EMVAccreditationSampleApp, which supports card present transactions with EMV cards (at `./EMVSampleApp/` in the SDK)
- SDKSampleApp, which supports card present transactions with mag stripe cards, card not present transactions with both types of cards, and PayPal transactions (at `./SDKSampleApp/`)

Both sample apps interoperate with a sample back-end server that is also distributed with the SDK.

Both sample apps have code for initializing the SDK with the credentials of a merchant that is pre-defined in the Sandbox. SDKSampleApp has this code in `STAppDelegate.m`; EMVAccreditationSampleApp has it in `OEMVAuthLoginViewController.m`.

You can run the sample apps in a simulator, but you must run them on a real iPhone to attach and use a card reader, and thus to perform transactions.

Note. SDKSampleApp initializes the SDK to run in the production environment by default. You can set the SDK to run in the sandbox by calling:

```
[PayPalHereSDK
        selectEnvironmentWithType:ePPHSDKServiceType_Sandbox];
```

See the `configureServers` method in `STOauthLoginViewController.m`.

Functionality of SDKSampleApp

The following tables identify the parts of the sample apps that perform each step of the workflow that is described in Chapter 3, The transaction workflow.

Setup steps

Workflow step	Sample app code
Initialize the SDK	**SDKSampleApp:** `didFinishLaunchingWithOptions` method in `STAppDelegate.m` **EMVSampleApp:**

Authenticate the merchant	**SDKSampleApp:** call to `PayPalHereSDK` class's `setActiveMerchant` method in `STOauthLoginViewController.m` **EMVSampleApp:**
Set the active merchant	
Set the merchant's location	**SDKSampleApp:** references to `PPHLocation` class in `SettingsViewController.m` **EMVSampleApp:**
Start monitoring the card reader	

A card transaction

Workflow step	Sample app code
Start an itemized transaction (invoice)	**SDKSampleApp:** `onchargePressed` method in `STTransactionViewController.m` **EMVSampleApp:**
Add items to the invoice	**SDKSampleApp:** `addItemWithId` method in ST`TransactionViewController.m` **EMVSampleApp:**
Take a payment using a credit card reader	**SDKSampleApp:** Call to `PPHTransactionManager` class's `processPaymentWithPaymentType` method in `PaymentMethodviewcontroller.m`, where the `ePPHPaymentMethodSwipe` enum is specified **EMVSampleApp:**
Finalize a payment with a customer signature	**SDKSampleApp:** `finalizePaymentForTransaction` method in `SignatureViewController.m`; also see the read-me file, `./README.md` **EMVSampleApp:**
Send a receipt	**SDKSampleApp:** `sendReceipt` method in `ReceiptInfoViewController.m` **EMVSampleApp:**

Variations on the basic workflow

Workflow step	Sample app code
A transaction without an invoice	**SDKSampleApp:** Call to `PPHTransactionManager` class's `beginPaymentWithAmount:andName` method in `TransactionViewController.m` **EMVSampleApp:**

A card not present transaction	**SDKSampleApp:** Call to `PPHTransactionManager` class's `processPaymentWithPaymentType` method in `ManualCardEntryViewController.m`, where the `ePPHPaymentMethodKey` enum is specified **EMVSampleApp:**
A check-in transaction	**SDKSampleApp:** `takePaymentUsingCheckinClient` method in `CheckedInCustomerViewController.m` **EMVSampleApp:**
Authorization and capture	**SDKSampleApp:** *Not implemented* **EMVSampleApp:**

Running the sample app

After you set up and run the sample server as described in **Error! Reference source not found.**, you can step through the code in one of the sample apps while you accept a test payment.

To use a sample app to accept a credit card, you must run the app on a physical device (rather than a similar), and you must attach a PayPal card reader to the device.

If a signature is required, the current SDK release requires capture of the signature before submission of a payment transaction; it is planned that a future release will not require this initial capture of a signature.

Appendix B. The sample server

The sample back-end server is a Node.js application. It is designed to interoperate with the sample app, and also to serve as a model for developing your own back-end server.

The sample server and app delivered with the SDK use a pre-defined Sandbox merchant, client ID, and secret. The merchant, named "teashop," is specified in `./sample-server/server.js`. The client ID and secret in are in `./sample-server/config.js`. You can replace all of these values with your own.

For background about what a back-end server does and how it works, review Overview of Your Back-End Server. Also review the code comments in these files:

- `./sample-server/config.js`
- `./sample-server/server.js`
- `./sample-server/lib/oauth.js`

Setting up the sample server

The sample server is located in the SDK distribution files at `./sample-server`.

The sample server is easily deployed on the Heroku web app hosting site, and is easily modified to run elsewhere. Run the sample server on a shared resource; one instance of it can serve multiple developers.

To prepare the sample server for use on a local development system or server:

1. Install Node.Js.
2. Start a terminal window, go to the `./sample-server` folder, and run `npm install`.
3. Run `node server.js`. The log messages are displayed in the terminal window. The server advertises itself using `Bonjour/zeroconf`, so the sample app should find it automatically. The Log In with PayPal return URL is not automated, but you configured it in Step 1, above.

Functionality of the sample server

The sample server implements both of the basic designs described in Developing a back-end server. To proxy all SDK operations to the server (the default), set

`exports.CENTRAL_SERVER_MODE` to true. To proxy only operations used to obtain access tokens, set it to false.

The sample server implements four URIs:

- `/login`: A dummy version of user authentication. Returns a ticket that can be used in place of a password to reassure you that the person you're getting future requests from is the person who entered their password in your app.
- `/goPayPal`: Validates the ticket and returns a URL which your app can open in Safari to start the Log In with PayPal (LIPP) flow. This method specifies the OAuth scopes you're interested in, which must include the PayPal Here scope (defined as https://uri.paypal.com/services/paypalhere openid email phone profile address https://uri.paypal.com/services/paypalattributes/business).
- `/goApp`: The endpoint to which PayPal returns the user after the user completes authentication. This endpoint inspects the result of authentication and redirects back to your app.

  First, the sample server sends a request to PayPal to exchange the refresh token for an access token:

```
javascript request.post(
    {
        url:config.PAYPAL_ACCESS_BASEURL +
        "auth/protocol/openidconnect/v1/tokenservice",
        auth:{ user:config.PAYPAL_APP_ID,
        pass:config.PAYPAL_SECRET, sendImmediately:true},
        form:{ grant_type:"authorization_code", code:req.query.code }
    },
    function (error, response, body) { }
);
```

  The server encrypts the access token received from PayPal using the ticket, so if someone hijacks your app's URL handler, the data is not usable; it is not the data sent by the LIPP flow. The server returns a URL to your app that allows the app to generate a refresh token when necessary. This URL is to the `/refresh/username/token` handler, and includes the refresh token issued by PayPal, encrypted with an account-specific server secret. The refresh token is never stored on the server, and is not stored in a directly-usable form on the client either. This minimizes the value of centralized data on your server, and allows you to cut off refresh tokens in cases of compromised tokens.

- `/refresh/username/token`: Decrypts the refresh token and passes it to the token service to get a new access token.

When you are ready to set up a server with your own client ID instead of the predefined client ID that the distributed samples use, set the return URL to point to your server. If you are testing on a real device, this URL generally needs to work on that device and on your simulator, meaning it must have a live DNS entry on the internet.

About the encrypted access token

The first time you run the sample app, the sample server sends it an encrypted access token and a refresh URL (a URL to the sample server that can be used to refresh the token).

The sample server encrypts the access token with a seed value which it calls a "ticket." The ticket is created when the user logs in to the server for authentication for the first time. This technique for acquiring a seed for encrypting the access token is suitable for use in live back-end servers, but other techniques are possible.

The `/sample-server/server.js` file has a default, test merchant named “teashop,” whose address is a confirmed address in teashop’s PayPal account. When you begin developing an app, your code can retrieve a merchant’s data (including the confirmed address, which is required for further operations) for an order (invoice) with the `PayPalHereSDK` class’s `activeMerchant` method. For more information, see the `PayPalHereSDK` class in the PayPal Here SDK’s reference documentation.