Developer Guide

PayPal Here SDK for Android

This document is confidential and its distribution is restricted.

DISTRIBUTE THIS DOCUMENT ONLY UNDER PROPER NDA.

Publication date:
01/12/15

PayPal

Document Revision History

Pub. Date	Description of changes
1/12/15	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**, or **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 app, **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 PayPal Here SDK

The github repository for PayPal Here SDK for iOS contains:

- Object code and other resources for the SDK (in `./sdk/PayPalHereSDK_1.0.aar`)
- Reference (Javadoc) documentation for the SDK (in `./sdk/docs/apidocs/`)
- The *PayPal Here SDK Android Developer Guide* (this document; in `./sdk/docs/dg_PPH-android-sdk.pdf`)
- Source code for two sample apps that demonstrate use of the SDK:
  - PayPalHereEMVSampleApp, which supports card present transactions with EMV cards (in `./PayPalHereEMVSampleApp/`)
  - PayPalHereSampleApp, which supports card present and card not present transactions with mag stripe cards, card not present transactions with both types of cards, and check-in transactions (in`./samples/PayPalHereSampleApp/`)
- Source code for a sample mid-tier server (in `./sample-server/`)

The PayPal Here SDK's role in PayPal's product family

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 *Authenticating SDK operations*.

What you'll need

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

- Android Phones which have version Andriod OS Gingerbear or higher (Andriod SDK version 10 or later)

- An integrated development environment (IDE) that supports development for the Android environment.
- 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
- For app authentication in production, a mid-tier server to store your app's secret

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 reader
- 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 Starting to work with the SDK.

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 and interfaces

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 `CardReaderManager` class, with hardware devices.

INVOICE
INVOICEITEM
: Interfaces that represent an invoice and an item in an invoice. Apps generally refer to these interfaces rather than the classes that implement them. The `DomainFactory` class has methods which construct objects of the appropriate class for an invoice, invoice item, etc., in any given case.

DOMAINFACTORY
: A "factory" class which creates objects that represent invoices, invoice items, and other transaction elements.

TRANSACTIONMANAGER
: An interface that defines a stateful transaction manager; takes payments and processes refunds on the current invoice (if any).

CARDREADERMANAGER
An interface that defines a card reader manager. Handles interaction with mag stripe and EMV credit card readers, including audio readers, dock port readers, and Bluetooth readers.

CARDREADERLISTENER
Translates between row card reader events and a delegate interface, Supports multiple listeners.

DEFAULTRESPONSEHANDLER
A generalized response handler for PayPal Here SDK methods that need one.

When you call such a method, you can provide a response handler to it by constructing an instance of `DefaultResponseHandler<param,err>`, where `param` is an object to be passed to the handler's `onSuccess` method, and `err` is an object (customarily an instance of `PPError`) to be passed to its `onError` method. You can customize the handler's behavior by overriding `onSuccess` and `onError`.

Chapter 2 Before you start

Obtaining the SDK

Before using the SDK, please contact us at DL-PayPal-Here-SDK@ebay.com to get the proper scope.

After downloading the SDK from the public GitHub, put the SDK's .aar file (at `./sdk/PayPalHereSDK_1.0.aar`) in an appropriate location for inclusion in your PayPal Here development project.

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.

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. You log in to the PayPal developer site and create a PayPal application. (Here "Create a PayPal application" means, "Get OAuth credentials for an app.") 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 mid-tier server, *not* in the app itself.

2. When a merchant begins using your app, your mid-tier server should direct them to PayPal's Log In with PayPal (PayPal Access) endpoint. 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 refer to an endpoint on your mid-tier server which PayPal can use to direct the merchant's device back to your app.

   An app should require a merchant to log in each time it is opened. Once a merchant has logged in, their credentials typically are considered good until until the merchant logs out or the app is opened again.

3. The Identity service prompts the merchant to grant your app permission to execute PayPal Here transactions on their behalf. The merchant grants

permission by entering their PayPal account's username and password and possibly answering questions about the scope of permission they want to grant. PayPal then redirects your mid-tier server to the return URL, returning the merchant's device to your app. If the merchant granted permission to execute transactions, the redirect includes an **authentication code** which represents the scope of the permission they granted.

4. Your app submits its client ID and the authentication code to the Identity service through the mid-tier server, and obtains a long-lived **refresh token**. (For security, the mid-tier server should encrypt the token. For more information see Developing a mid-tier server and The sample server.)
5. Your app submits the refresh token to the Identity service (again through the mid-tier server) to obtain a short-lived **access token**. This step must be repeated periodically when the access token expires.
6. Your app must include the current access token in each call it makes to the PayPal Here SDK.
7. To handle token expiration and obtain a new token, implement the AuthenticationListener interface and define the onInvalidToken method. Typically the onInvalidToken method invokes the previously-obtained refresh URL to retrieve a new access token, refresh URL, and token expiration value. For additional information, see Payments with the Sample App.

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, and the sample server's source code.

How to provide OAuth credentials to the SDK

Construct an `OauthCredentials` object, providing the access token to the constructor. Store the new object in a `Credentials` variable. Give the credentials to the SDK by calling `PayPalHereSDK.setCredentials()`:

```
String accessToken = . . .;
Credentials credentials = new OAuthCredentials(accessToken);
PayPalHereSDK.setCredentials(credentials,
        new DefaultResponseHandler<merchant, PPError<MerchantManager.merchantErrors>>() {
            @Override
```

```
            public void onSuccess(Merchant merchant) {
                // Perform any necessary setup, e.g., register an authentication listener.
                . . .
            }
            @Override
            public void onError(PPError<MerchantManager.merchantErrors> errors) {
                // Handle an error.
                . . .
            }
        }
);
```

How to refresh the access token

Because the access token is short-lived, your app needs to provide for refreshing it when it expires.

Implement the `AuthenticationListener` interface and define the `onInvalidToken` method to call the refresh URL. The refresh URL returns a new access token, refresh URL, and token expiration value. You need not save the new access token; the SDK does that internally.

```
import com.paypal.merchant.sdk.AuthenticationListener;
. . .
protected AuthenticationListener authListener = new AuthenticationListener() {
    @Override
    public void onInvalidToken() {
        String refreshUrl = . . .;
        RefreshTokenTask refreshAccessTokenTask = new RefreshTokenTask();
        refreshAccessTokenTask.execute(refreshUrl);
    }
};
```

Register the listener with the SDK:

```
PayPalHereSDK.registerauthenticationListener(authListener);
```

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 mid-tier 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 mid-tier server to interoperate with your app.

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

There are two basic approaches to designing a mid-tier server:

- You proxy all calls to the PayPal Here SDK (including retrieving tabs, retrieving locations, making payments, etc.) through the mid-tier server. The access token is stored on the server.
- You only proxy the operations used to obtain access tokens through the mid-tier 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 mid-tier server's functions are simple, the sample server distributed with the SDK is a suitable (and recommended) starting point for developing your own please see The sample server, for more information.

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.

Chapter 3 Starting to work with the SDK

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.

Common steps for taking payment

The steps in setting up an app are:

1. Initialize the SDK (each time the app starts)
2. Authenticate the merchant and pass the merchant's credentials to the SDK (the first time the merchant uses the app)
3. Determine the types of payment the merchant can accept (optional, may be useful depending on the app's logic)
4. 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)
5. Start monitoring the card reader for events (for card present transactions)

Initialize the SDK

You initialize the PayPal Here SDK by calling the class method `PayPalHereSDK.init`:

```
Public class LoginScreenActivity extends Activity {
   . . .
   PayPalHereSDK.init(getApplicationContext(), PayPalHereSDK.Sandbox);
```

The first parameter is the context of the global `Application` object. This example obtains it by defining a class that extends `Activity` and calling its `getApplicationContext` method.

The second parameter identifies the environment that the SDK is to operate in. It may be `PayPalHereSDK.Live` or `PayPalHereSDK.Sandbox`.

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 mid-tier server.

Set the active merchant

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

```
Credentials credentials = . . .;        // The merchant's OAuth credentials.
Final DefaultResponseHandler =          // A default response handler.
        new DefaultResponseHandler< Merchant, PPError<MerchantManager.MerchantErrors> >;
PayPalHereSDK.setCredentials(credentials, defaultResponseHandler);
```

Determine types of payment the merchant can accept

After authenticating the merchant, but before taking payment, you optionally can ask the SDK:

- Whether the logged-in merchant is allowed to take payments
- Whether the logged-in merchant is allowed to process check-in transactions, and the types of payment the merchant can take for them

You determine the types of payment that the merchant can accept by examining their `Merchant` object, as in this example method:

```
private void displayMerchantInfo() {
    Merchant m = PayPalHereSDK.getMerchantManager().getActiveMerchant();
    String email = "";
    String businessName = "";
    String currencyCode = "";
    // Get basic data about the logged-in merchant from the SDK's Merchant object.
    if (m!=null) {
        email = m.getEmail();
        businessName = m.getBusinessName();
        Currency c = m.getMerchantCurrency();
        If (c!=null) {
            currencyCode = c.getCurrencyCode();
    }
    // Determine whether the logged-in merchant can process transactions.
    String status = m.getMerchantStatus();
    // Determine the types of payment the logged-in merchant can take
    List<Merchant.AvailablePaymentTypes> paymentTypes = m.getAllowedPaymentTypes();
}
```

Set the merchant's location

In the `PayPalMerchantCheckinActivity.java` sample file, see the code that uses the `MerchantManager` 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. The SDK needs location information to take credit card payments and check-in payments. However, it isn't mandatory for a merchant to be checked in to take a credit card payment. 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

To start monitoring the card reader for a card swipe, call the card reader manager's `beginMonitoring` method:

```
PayPalHereSDK.getCardReaderManager().beginMonitoring(
        CardReaderListener.ReaderConnectionTypes.Bluetooth.
                CardReaderListener.ReaderConnectionTypes.AudioJack);
```

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)

A `TransactionManager` object represents a transaction. The SDK automatically creates a `TransactionManager` object for your use. You can fetch it at any time by calling `PayPalHereSDK.getTransactionManager()`

Once you have the `TransactionManager` object, call its `beginPayment` method:

```
TransactionManager transactionMgr = PayPalHereSDK.getTransactionManager();
.
.
.
Invoice mInvoice = transactionMgr.beginPayment();
```

For an example, see the `initInvoice` method in the `ItemizedActivity.java` sample file.

Add items to the invoice

Once you have created an invoice you can add items to it. An invoice item is represented by an `InvoiceItem` object. You can create one by calling `DomainFactory.newInvoiceItem()`:

```
String mItem = "Self-deploying umbrella";
BigDecimal mPrice = BigDecimal("149.95");
InvoiceItem mInvoice = DomainFactory.newInvoiceItem(mItem, mPrice);
```

For amounts that are entered for fixed-amount payments, or for items in an invoice, use double quotes ("") in the BigDecimal constructor to maintain the accuracy and precision of the entered amount.

`newInvoiceItem` accepts several combinations of parameters which let you specify information such as a tax rate and an inventory ID. See the reference documentation for `DomainFactory` for more information.

Once you have created an invoice item, you add it to the invoice by calling the `Invoice` object's `addItem()` method;

```
long quantityToAdd = . . .;
mInvoice.addItem(mItem, quantityToAdd);
```

The basic process of creating an invoice is described in Basic workflow for a transaction. This section provides some background and additional detail.

An invoice is represented by the `Invoice` interface. You create an invoice by calling `TransactionManager.beginPayment()`. You create invoice items by calling `DomainFactory.newInvoiceItem()` and add them to the invoice by calling `Invoice.addItem()`.

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.

In most locales you must set a tax rate for each item in an invoice. See the reference documentation for InvoiceItem.setTaxRate() and Invoice.setTaxEnabled(). Round tax amounts to the nearest penny (to two digits after the decimal point) at the item level.

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

Working with multiple invoices

An app can process several invoices concurrently, e.g. for multiple customers.

After you create an invoice by calling `TransactionManager.beginPayment()`, you can save a reference to it by calling the `TransactionManager` object's `getActiveInvoice()` method. Then you can create another invoice.

```
Invoice oldInvoice = transactionMgr.getActiveInvoice();   // Save current invoice
Invoice newInvoice = transactionMgr.beginPayment();       // Create new invoice
```

Once you have created two or more invoices, you can then switch between them at will. Call the `TransactionManager` object's `getActiveInvoice` method to preserve the current invoice, then call its `setActiveInvoice` method to activate another.

```
Invoice tInvoice   = transactionMgr.getActiveInvoice();   // Save current invoice
transactionMgr.setActiveInvoice(oldInvoice);              // Active old invoice
oldInvoice = tInvoice;
```

Finalize a payment

When an invoice is complete you must finalize it. Finalization encompasses all of the steps necessary to complete the invoice and get payment.

First, call the `TransactionManager object`'s `setInvoice` method to identify the invoice with the transaction:

```
transactionMgr.setInvoice(mInvoice);
```

Next, call the `TransactionManager object`'s `processPayment` method. The form of this call depends on the type of card being used:

```
transactionMgr.processPayment(PaymentType.CARD_READER, NULL,                    For mag stripe card
                              mResponseHandler);
transactionMgr.processPayment(activity, mResponseHandler);                      For EMV card
```

The call to `processPayment` is asynchronous, and so uses a response handler. The response handler receives a `TransactionManager.PaymentResponse` object if the operation succeeds (indicating that payment has been made), or a `PPError` object if the operation fails. Do not go on to the next step until the operation has succeeded.

Finally, for a mag stripe card transaction only, call the `TransactionManager` object's `finalizePayment` method:

```
transactionMgr.finalizePayment(PaymentType.CARD_READER, mResponseHandler);`For mag stripe card only
```

The call to `finalizePayment()` is also asynchronous. The response handler receives the same type of object as the `processPayment` response handler when the operation succeeds or fails.

Send the customer a receipt

After a mag stripe transaction is successfully completed, the merchant can send the customer a receipt by calling the `sendReceipt` method.

```
```

This step applies to mag stripe transactions only. For EMV transactions the SDK sends the customer a receipt automatically.

The receipt may be delivered by email or by phone, depending on what contact information the customer gave.

Variations on the basic workflow

This section describes several variations on the basic workflow above.

A fixed-amount transaction

A **fixed-amount transaction** is one that specifies a fixed amount of payment due. Such a transaction has no invoice.

To start a fixed-amount transaction, call `TransactionManager.beginPayment()` with the fixed amount as a parameter:

```
BigDecimal mFixedAmount = . . .;
transactionMgr.beginPayment(mFixedAmount);
```

In addition to the fixed amount, you must specify the mode of payment (e.g., card reader) and a callback handler that will be notified of the transaction status.

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).

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.

To pay an invoice with the result of a card swipe, you must first gather the signature image. The `PPHSignatureView` can be placed in a view controller of your own design and it will provide an image which can be sent to the API.

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

A card not present transaction

To start a **card not present transaction,** create a ManualEntryCardData object and set the card's card number, expiration date, CCV2, and card holder name.

```
private String
    cardNumber = . . .,
    expDate = . . .,
    ccv2 = . . .,
    cardholder = . . .;
ManualEntryCardData manualEntry =
        DomainFactory.newManualEntryCardData(cardNumber, expDate, ccv2);
manuLEntry.setCardHoldersName = cardholder;
```

Complete the transaction in the same way as calling processPayment for authorize and capture transaction.

A check-in transaction

A transaction that is paid from a PayPal account rather than a card account is called a **check-in transaction** because it uses a workflow in which a customer creates a tab by checking in to a merchant location. For a more complete summary of the check-in transaction workflow, see Chapter 1, Overview. For detailed information about check-in transactions, see the Before you start a check-in transaction, the customer must be checked in to the merchant location.

First you must get a list of checked-in customers. Create a `List` object to hold the list of checked-in customers; then call `MerchantManager.getCheckedInClientsList()`. That method expects one parameter, a `DefaultResponseHandler` whose onSuccess method receives the list of checked-in customers.

```
MerchantManger merchantMgr = PayPalHereSDK.getMerchantManager();
merchantMgr.getCheckedInClient(
        new DefaultResponseHandler<
                List<CheckedInClient>,
                PPError{MerchantManager.merchantErrors>
        >() {
            @Override
            public void onSuccess(List<CheckedInClient> checkedInList) {
                // Display list of checked-in customers and invite the cashier to select one.
                . . .
            }
            @Override
            public void onError(PPError<MerchantManager.merchantErrors> errors) {
                // Handle an error.
```

```
                . . .
            }
        }
}
```

Display the list to the cashier and invite them to select the checked-in customer who is paying. In the code above, this would be done in `onSuccess()`.

Complete the transaction using the same procedure as calling processPaymet for authorize and capture transaction.

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, set card data. and define an invoice in the usual way.
2. Instead of taking payment in the usual way, call the transaction manager's `authorizePayment` method, passing a payment type and a response handler. (The SDK supports several forms of `authorizePayment()`; see the reference documentation at `./docs/apidocs/index.html` in the SDK for details.)

```
TransactionRecord mRecord;
manager.authorizePayment(
    TransactionManager.PaymentType.CardReader,
    new DefaultResponseHandler<
        TransactionManager.PaymentResponse,
        PPError<TransactionManager.PaymentErrors>
    >() {
        @Override
        public void onSuccess(TransactionManager.PaymentResponse response) {
            mRecord = response.getTransactionRecord();
        }
        @Override
```

```
        public void onError(PPError<TransactionManager.PaymentErrors> e) {
            // Error out
        }
    }
);
```

3. If the response handler calls `onSuccess`, call the response object's `getTransactionRecord` method. It returns a `TransactionRecord`, which you should save. You can get information about the transaction from this object.

Capturing a payment

To **capture** an authorized payment, call the transaction manager's `capturePayment` method, passing the `TransactionRecord` returned by `authorizePayment` and a response handler.

```
manager.capturePayment(
    mRecord,
    new DefaultResponseHandler<
        TransactionManager.PaymentResponse,
        PPError<TransactionManager.PaymentErrors>
    >() {
        @Override
        public void onSuccess(TransactionManager.PaymentResponse response) {
            // Do something
        }
        @Override
        public void onError(PPError<TransactionManager.PaymentErrors> e) {
            // Error out
        }
    }
);
```

The amount captured may exceed the original authorized amount by a specified percentage to allow for 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 `doVoid` method, passing the `TransactionRecord` and a response handler:

```
manager.voidAuthorization(
    mRecord,
    new DefaultResponseHandler<
        TransactionManager.PaymentResponse,
        PPError<TransactionManager.PaymentErrors>
    >() {
        @Override
```

```
        public void onSuccess(TransactionManager.PaymentResponse response) {
            // Do something
        }
        @Override
        public void onError(PPError<TransactionManager.PaymentErrors> e) {
            // Error out
        }
    }
);
```

Refunding a captured payment

To **refund** a payment, call the transaction manager's `doRefund` method, passing the `TransactionRecord`, the amount to refund, and a response handler:

```
manager.doRefund(
    mRecord,
    "1.00",
    new DefaultResponseHandler<
        TransactionManager.PaymentResponse,
        PPError<TransactionManager.PaymentErrors>
    >() {
        @Override
        public void onSuccess(TransactionManager.PaymentResponse response) {
            // Do something
        }
        @Override
        public void onError(PPError<TransactionManager.PaymentErrors> e) {
            // Error out
        }
    }
);
```

To refund the entire amount of the payment, set the second parameter to null.

Additional capabilities

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

Refunding a payment

Mag stripe transactions

To refund a payment, call `TransactionManager.doRefund()`:

```
manager.doRefund(
```

```
    mRecord,
    BigDecimal("1.00"),
    new DefaultResponseHandler<
        TransactionManager.PaymentResponse,
        PPError<TransactionManager.PaymentErrors>
    >() {
        @Override
        public void onSuccess(TransactionManager.PaymentResponse response) {
            // Do something
        }
        @Override
        public void onError(PPError<TransactionManager.PaymentErrors> e) {
            // Error out
        }
    }
);
```

In this example, `mRecord` is the `transactionRecord` that was returned when the payment was made. The second parameter is the amount of the refund, which must be a `BigDecimal`. The third parameter is the response handler. A successful call returns a new `transactionRecord`.

EMV transactions

To refund a payment, call `TransactionManager.doRefund()`:

```
manager.doRefund(
    mRecord,
    BigDecimal("1.00"),
    new DefaultResponseHandler<
        TransactionManager.PaymentResponse,
        PPError<TransactionManager.PaymentErrors>
    >() {
        @Override
        public void onSuccess(TransactionManager.PaymentResponse response) {
            // Do something
        }
        @Override
        public void onError(PPError<TransactionManager.PaymentErrors> e) {
            // Error out
        }
    }
);
```

In this example, mRecord is the `transactionRecord` that was returned when the payment was made. The second parameter is the original invoice. The third parameter is the response handler. A successful call returns a new transaction record.

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 referrer codes from your account manager.

If each instance of your app uses a single referrer code, you can set the referrer code at the SDK level. Every invoice your app creates will bear the same referrer code. Call `PayPalHereSDK.setReferrerCode()` during app setup:

```
PayPalHereSDK.setReferrerCode(mReferrer);
```

The parameter must be a `String` that contains the referrer code.

If an instance of your app uses different referrer codes for different transactions, you must set the referrer code at the invoice level. Call the `Invoice` object's `setReferrerCode` method while you are building the invoice:

```
mInvoice.setReferrerCode(mReferrer);
```

Again, the parameter must be a `String` that contains the referrer code.

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, call the `Invoice` object's `setCashierId` method while you are building the invoice:

```
mInvoice.setCashierId(cashier_ID);
```

The parameter must be a `String` that contains the 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 `getAllowedPaymentTypes` method of the `Merchant` class to determine whether the logged-in merchant is approved for PayPal Here.

Example Use Case: Credit Card Reader

After an app initializes the SDK, the app can enable a merchant to take a payment with a credit card reader. The app could use the following steps:

1. Obtain an instance of `TransactionManager` and register a `TransactionListener` object to be notified of status messages related to the payment. The `TransactionManager` interface is stateful and thus saves data between calls.
2. Access the `CardReaderManager` interface and register a `CardReaderListener` object to be notified of status messages related to the card reader device.
3. Create a transaction, or "summary of charges," to present to the customer.
4. Initiate a transaction by calling `TransactionManager.beginPayment()`. If you specify an amount, the payment is for a fixed amount. If you do not specify an amount, the transaction is based on an invoice.
5. If necessary, prompt the user to connect a card reader. Use the `CardReaderListener` object to tell the user when a card can be swiped or inserted. Also notify the user when transaction processing begins, and if necessary, prompt for a signature.
6. Notify the user when the transaction has been processed.

If your app uses a custom credit card reader, or if you want to otherwise **customize the SDK**, see Customizing the SDK with a Transaction Controller.

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 create a `ManualCardEntry` object by calling `DomainFactory.newManualEntryCardData()`:

```
ManualCardEntry mManualCard =
        DomainFactory.newManualEntryCardData(cardNumber, expirationDate, cvv2);
```

Each of the parameters is a `String`.

Validating the card data

Your app should validate the card number, expiration date, and CVV2 before providing the `ManualCardEntry` object to the SDK.

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

The `expirationDate` must be in the form *mmyyyy*, e.g., `072017` for July 2017, and should represent the current month or a future month.

The CVV2 must be a three-digit number.

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

Presenting the card number and date to the user

Your app's user interface should automatically present the card number in groups of digits, as is customarily done for the account number printed on a card. American Express cards, whose **bank identification numbers** (BINs) start with 34 or 37, have 15 digits, should be present in groups of 4, 6, and 5 digits. Most other card numbers have 16 digits, and should be present in four groups of 4 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.

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 `TransactionManager` class to take credit card payments *without* performing this type of detailed communication.

Starting Communication with the Card Reader

The `CardReaderManager` class handles interactions 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 listening for a connection to a card reader. You do this by defining a class which extends `TransactionListener` and `CardReaderListener`, then register the listener with the card reader manager:

```
public class CreditCardPeripheralActivity extends MyActivity implements TransactionListener,
CardReaderListener {
    @Override
    public void onResume() {
        super.onResume();
        registerTransactionAndCardReaderListener(true);
        . . .
    }

    @Override
    public void onPause() {
        super.onResume();
        registerTransactionAndCardReaderListener(false);
        . . .
    }

    private void registerTransactionAndCardReaderListener(Boolean isRegister) {
        if (isRegister) {
            PayPalHereSDK.getTransactionManager.registerListener(this);
            PayPalHereSDK.getCardReaderManager.registerListener(this);
         }
        else {
            PayPalHereSDK.getTransactionManager.unregisterListener(this);
            PayPalHereSDK.getCardReaderManager.unregisterListener(this);
        }
    }
}
```

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 Javadoc description of `PPHCardReaderWatcher` for details.

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

Location data

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:

7. 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.

8. In operation, periodically retrieve customers who have checked in (who have opened tabs) at the location.
9. Allow the merchant to select a customer who wants to pay their tab.
10. Create an invoice for the customer, select items for the invoice, and confirm the invoice amount.
11. 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 CheckedInClient 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 `MerchantManager` class's `getCheckedInClientsList` method.

The following code gets a list of locations and displays the all the checked in clients.

```
[
PayPalHereSDK.getMerchantManager().getCheckedInClientsList(
                  new DefaultResponseHandler<List<CheckedInClient>,PPError<MerchantErrors>> {

                   public void onSuccess(List<CheckedInClient> list) {
                      for(CheckInClient client: list){
                         Log.d("Location"," ID: "+client.getCheckInId();
                      }
                  });
```

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 Customizing the SDK with a transaction controller

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

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

The `TransactionController` interface 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:

- `onPreAuthorize`. Invoked by the SDK right before the SDK calls the Mobile In-Store Payments API, for taking a payment (authorizing a payment). The Invoice object (in `inv`) 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 `TransactionController` interface, or
- Implement the `TransactionController` interface, and in the `onPreAuthorize` method, return the `TransactionControlAction.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 `preAuthJSON` 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 `MyActivity.java` sample file.

Chapter 7 Handling errors

When the PayPal Here SDK encounters an error it returns a `PPError` object. This object has several properties that provide potentially useful information about the error.

The following sections explain how your app can recognize and respond to various types of errors.

For information about specific errors and recommended responses to them, see the reference documentation for the PayPal Here SDK for Android. Clone the `PayPal-Mobile/android-here-sdk-dist` repository from https://github.com, and load the reference documentation directory page from `./html/index.html`.

Error processing best practices

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
- Log the error, tell the user that something has gone wrong, and abandon the operation that the app is trying to perform (or some part of it)
- 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 its user experience will be.

Error handling classes

The `PPError` class

The `PPError` class represents an error. A DefaultResponseHandler object expects a PPError object as its second parameter; when the object's onError method is called, the PPError object provides information about the error that occurred.

`PPError has two public methos:`

- `getDetailedMessage()` returns a `String` that describes the error in terms that should be meaningful to a user of an app.
- getErrorCode() returns an enum value of type T which represents the type of error that occurred.

The `PPError.BasicErrors` class

The `PPError.BasicErrors` class represents a basic type of error. It has two methods:

- `valueOf()` returns a String that indicates a basic type of error. Its value can be `"NotYetImplemented"`, `"Success"`, and `"Unknown"`.
- `values()` returns an array of Strings that includes all of the basic types of errors.

Appendix A. The sample apps

Functionality of sample apps

The following tables identify the parts of the sample apps that perform each step of the workflow described in Starting to work with the SDK.

Setup steps

Workflow step	Sample app code
Initialize the SDK	**PayPalHereSampleApp:** `PayPalHereSDK.init()` call in `LogInActivity.onCreate()` **PayPalEMVSampleApp:**
Authenticate the merchant	**PayPalHereSampleApp:** **PayPalEMVSampleApp:**
Set the active merchant	**PayPalHereSampleApp:** `setCredentials()` call in `OAuthLoginActivity.setMerchantAndCheckin` **PayPalEMVSampleApp:**
Set the merchant's location	**PayPalHereSampleApp:** `OAuthLoginActivity.getMerchantManager` method **PayPalEMVSampleApp:**
Start monitoring the card reader	**PayPalHereSampleApp:** **PayPalEMVSampleApp:**

Steps in the basic workflow

Workflow step	Sample app code
Start an itemized transaction (invoice)	**PayPalHereSampleApp:** `ItemizedActivity.initInvoice()` **PayPalEMVSampleApp:** `MainActivity.java`
Add items to the invoice	**PayPalHereSampleApp:** `ItemizedActivity.addFruit()` **PayPalEMVSampleApp:**
Finalize a payment	**PayPalHereSampleApp:** `CreditCardPeripheralActivity.finalizePayment()` (a private method which calls the `TransactionManager's finalizePayment` method) **PayPalEMVSampleApp:** n/a

Finalize a payment with a customer signature	**PayPalHereSampleApp:** **PayPalEMVSampleApp:**
Send a receipt	**PayPalHereSampleApp:** `CreditCardPeripheralActivity.java` **PayPalEMVSampleApp:** n/a

Variations on the basic workflow

Workflow step	Sample app code
A fixed-amount	**PayPalHereSampleApp:** `OnClickListener.onclick()` in `FixedPriceActivity.java` **PayPalEMVSampleApp:**
A card not present transaction	**PayPalHereSampleApp:** `CreditCardManualActivity.takePayment()` **PayPalEMVSampleApp:** n/a
A check-in transaction	**PayPalHereSampleApp:** Get list of checked-in customers: `PayPalMerchantCheckinActivity. getCheckedInClientList()`; display list: `PayPalMerchantCheckinActivity. displayClientList(); take payment: PayPalMerchantCheckinActivity.takePaymentWithCheckedInClient()` **PayPalEMVSampleApp:** n/a
Authorization and capture	**PayPalHereSampleApp:** **PayPalEMVSampleApp:**

Additional capabilities

Workflow step	Sample app code
Refund a payment	**PayPalHereSampleApp:** `CreditCardPeripheralActivity.doRefund()` **PayPalEMVSampleApp:** `SalesActivity.performRefund()`

Notes on the sample apps

Authentication

The mid-tier server provides the ticket ID and merchant information (username, business name, address, etc.) to the sample app.

After the sample app has retrieved the access token, the app creates a `Credentials` object and sends it to a `PayPalHereSDK` object. This prepares your app's `MerchantManager` object

to accept payments. See the `OAuthLoginActivity.setMerchantAndCheckIn()` method in the sample app's `OAuthLoginActivity.java` file.

Merchant location

The sample app sets the merchant's location automatically when the merchant logs in. Although your app can separately provide an option to "check in" a merchant, the sample app checks in the merchant right after the merchant successfully logs in.

The setup process

The sample apps work with a testing server, which is an instance of the sample server hosted by PayPal. Thus you need not set up and run the sample server to run the sample apps.

The first time you run a sample app it prompts you to log in to the testing server with the server's test credentials. Enter the username `teashop` and the password `11111111` (eight numeral 1's). The server provides ticket ID and merchant information (username, business name, address, etc.) to the sample app.

After you log in you are redirected to PayPal's log-in page, where you must log in with a PayPal merchant account's username and password. This links `teashop`'s PayPal account to the sample app's app ID.

After the PayPal account is linked to the app ID you are returned to the return URL in the testing server. The testing server completes authentication and then redirects you to the sample app. This enables the sample app call the PayPal Here API directly through the SDK.

Running the sample apps

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 simulator), and you must attach a PayPal card reader to the device.

The first time you run the sample app you must log in to the testing server with merchant username `teashop` and password `11111111` (eight number 1's). After you log in you are redirected to PayPal's log-in page, where you must log in with a PayPal merchant account's username and password. Once you log in to PayPal the sample app completes its setup as descrbied in The setup process.

Appendix B. The sample server

The sample mid-tier 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 mid-tier server.

The sample apps distributed with the SDK are configured to work with an instance of the sample server, called the "testing server," that is hosted by PayPal. Thus you need not set up and run the sample server in order to use the sample apps. You will need to set up and customize the sample server when you are ready to start testing your own app, though.

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

For background about what a mid-tier server does and how it works, review Developing a mid-tier server. Also review the code comments in these files:

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

Setting up the sample server

The sample server is located in the SDK distribution files at `./sample-server/heroku-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`. 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 mid-tier 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`).
- `/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.

The merchant's username and password are stored in a database defined at `./sample_server/heroku_server/users.db`. (This is an nStore database; nStore is a database for Node.js applications.) The access and refresh tokens are stored in the same database when they become available.

For additional information, see *Developing a mid-tier server* and the source code comments in `./sample_server/heroku_server/server.js`.

About the encrypted access token

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

The testing 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 mid-tier servers, but other techniques are possible.

The sample server distributed with the SDK handles the access token the same way as the testing server. (Recall that the testing server is actually an instance of the sample server hosted by PayPal.)

The `/sample-server/heroku-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.