Merchant Aggregator PG

Onboarding Guide

For In-App Payments and Account Preprovisioning

US Edition | April 2019 | Samsung Pay SDK 2.6.00 | Doc Rev 1.0

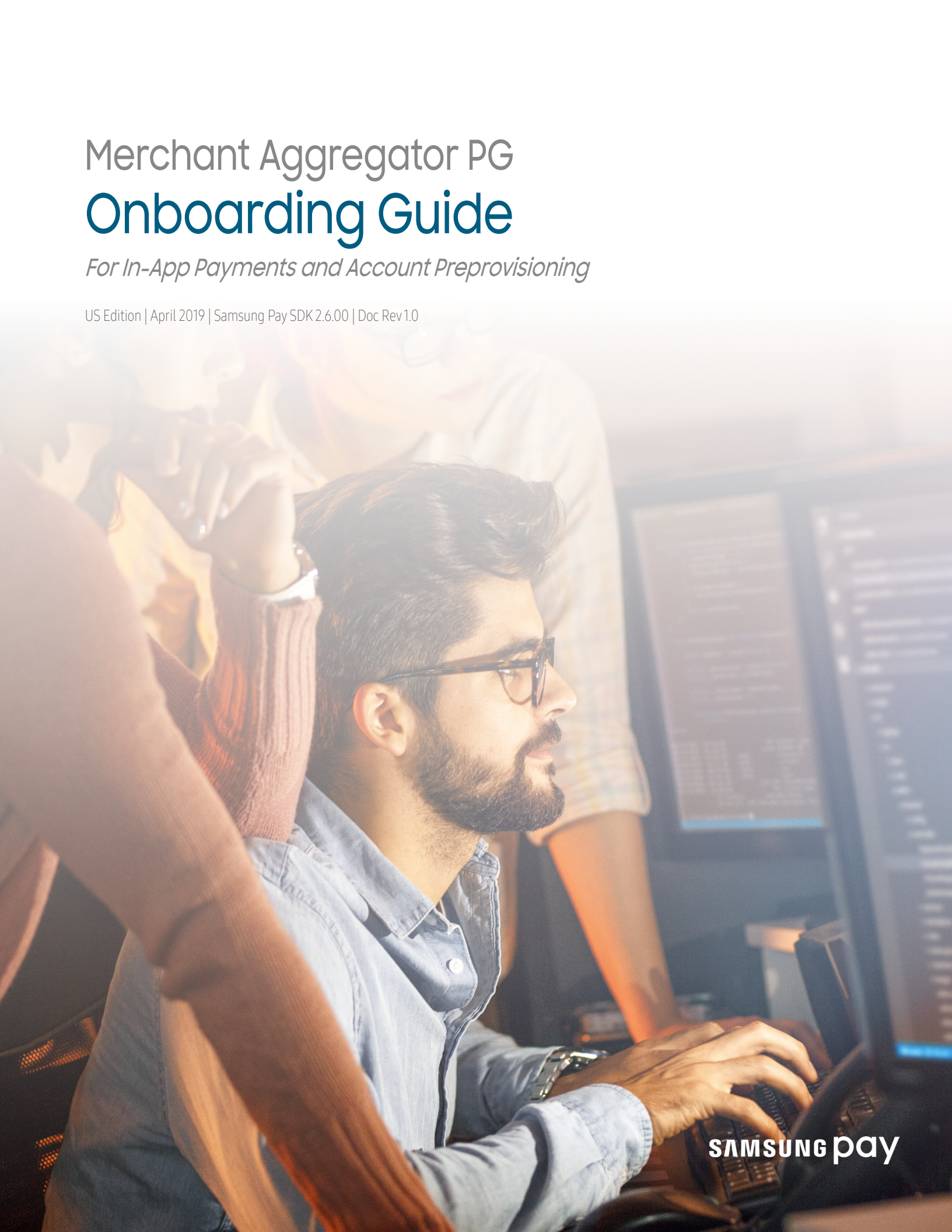

Table of Contents

List of Illustrations

Samsung Pay and the PG – what's the relationship?

Accepted almost anywhere a credit card can be swiped or tapped, Samsung Pay also works in-app and online. For Samsung Pay users purchasing goods and services online, it is our payment gateway (PG) partners that actually process the credit card payment, authorizing transactions upon electronic approval by the card issuer. In terms of online payments, PGs help expand the merchant's business by improving customer service, consolidating vendors, and streamlining reporting and reconciliation.

Moreover, when it comes to integrating merchant apps with Samsung Pay, the PG can serve as a merchant "aggregator," doing most of the heavy lifting via extensions to its proprietary SDK and, in the process, save its merchant partners significant development time and technical resources. Or, if they prefer, merchants can onboard with Samsung Pay and the PG separately and then undertake the required integration work independently. The latter approach generally applies to merchant apps that are already successfully integrated with their PG's SDK and want to avoid disruption.

This document offers guidance on the merchant aggregation option, strongly advocating this approach to (a) standardize the merchant onboarding experience by minimizing the necessary touchpoints and (b) reduce the go-to-market (GTM) time typically required for merchant app deployments that support Samsung Pay as a payment method.

A PG-Samsung Pay partnership:

- Allows merchants to easily support Samsung Pay as an online payment method from their PG-enabled Android applications
- Provides simple configurations for the majority of merchant use cases
- Supports automated new customer provisioning flows in addition to traditional checkout.

Keeping the foregoing in mind, let's take brief look at the Samsung Pay SDK and then we'll step through the respective onboarding flows.

What is the Samsung Pay SDK and what does it do?

The Samsung Pay SDK is an application framework for integrating selected Samsung Pay features with Android-based partner apps on Samsung devices. For merchant apps, Samsung Pay's **In-App Payments service** gives online customers the option of paying for products and services with Samsung Pay, as well as letting merchants set up (provision) a new customer account on demand, essentially auto-filling the merchant's online account creation form with corresponding information from Samsung Pay. Samsung Pay's trusted authentication mechanisms (PIN, iris or fingerprint scan) ensure that both payment and any user profile information shared with the merchant are authorized by the legitimate Samsung Pay user.

Before diving deeper into the Samsung Pay SDK and its constituent components, it's important to understand how the in-app payments service works. Such a discussion begins on the next page.

Understanding the In-App Payments service

First of all, a mobile wallet like Samsung Pay gives users the freedom to securely enroll and store their credit, debit, and gift cards on their Samsung Galaxy smartphones and Samsung wearables. Samsung Pay replaces the actual card number with a unique digital card number called a token so the true card number is never revealed and therefore cannot be compromised. In other words, users can leave their bulky and often disorganized billfolds loaded with plastic cards at home and, instead, use their smartphones and wearables to make purchases with the added confidence that their card information is protected. And now the technology that supports secure in-store purchases can be used for online (in-app) payments right from the merchant app.

Depending upon the PG, Samsung Pay's In-App Payments service sends either (a) an encrypted network token that allows the merchant or its PG to decrypt the payload and process the payment or (b) a PG token (also called a gateway token) that lets the merchant handle the payload in much the same way as normal card payments are handled by their app.

But that's not all. To help streamline the overall online shopping and payment experience, Samsung Pay APIs can be called to obtain the user profile information needed to create a new user account with the merchant's online store — name, billing address, shipping address, phone, email, or any combination thereof — saving the user redundant data entry. Like requesting payment, account preprovisioning requires secure user authentication before any personal information is shared with the merchant app. If users have items in their shopping carts and are ready to checkout, they can then select Samsung Pay as the payment method.

The most important prerequisite for both payment and account preprovisioning is that both apps — (a) Samsung Pay and (b) the merchant app integrated with the Samsung Pay SDK and the PG's SDK — are installed on the same eligible Samsung mobile device. You can review the list of Samsung Pay-compatible devices at https://www.samsung.com/us/support/owners/app/samsung-pay under **Compatiblity**.

Tokenized payments

Samsung Pay's In-App Payments service supports two types of tokens — gateway tokens (indirect) and network tokens (direct). What's the difference? Well, if you're a PG like First Data that can accept encrypted token bundles *directly* from the user's device over the internet, the merchant app will need to request an encrypted token bundle from Samsung Pay first, before sending it on to the PG. By contrast, if you're a PG like Stripe, the transaction authorization request is sent from the SDK-integrated merchant app to the PG via the Samsung-PG Interface Server (PGIS), which furnishes a gateway token, along with merchant identification.

Figure 1 illustrates the general process flow between the merchant app, Samsung Pay, and the financial network, including the PG.

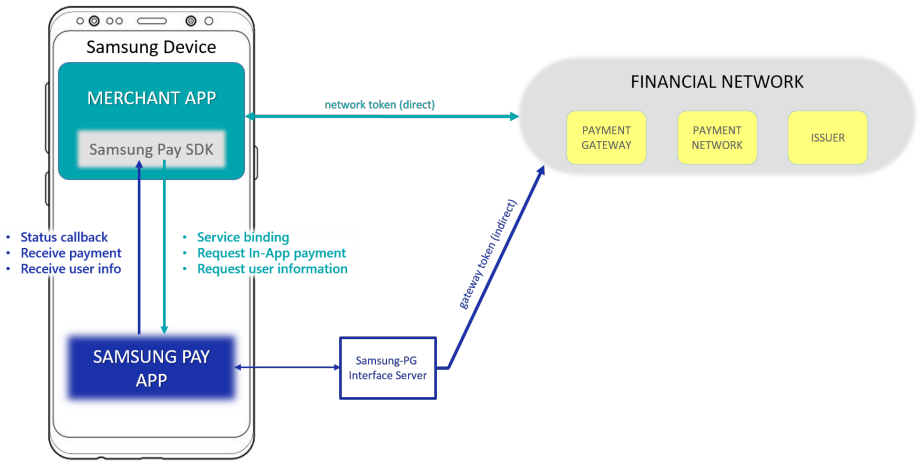

Figure 1. General in-app payment process flow

The essential difference is that a network token requires the merchant app to handle decryption of the token bundle or work with the PG to handle decryption, whereas gateway token decryption is always handled by the PG via the Samsung-PG Interface Server. The essential differences in terms of process steps are enumerated in Figures 2a and 2b, respectively.

Network Token mode (direct)

1. In the merchant app at checkout, the user selects Samsung Pay as the payment method; the Samsung Pay app requests partner verification from the Samsung Pay Online Payment Server.
2. Encrypted payment information is passed from the Samsung Pay app to the PG through the merchant app via the PG's SDK and governing APIs.
3. Applying the merchant's private key, the PG decrypts the payment information structure and processes the payment through the Acquirer and the appropriate payment card network (American Express, Discover, Mastercard, or Visa).
4. Upon receiving the card issuer's verdict (accept/decline), the PG notifies the merchant app via applicable APIs.

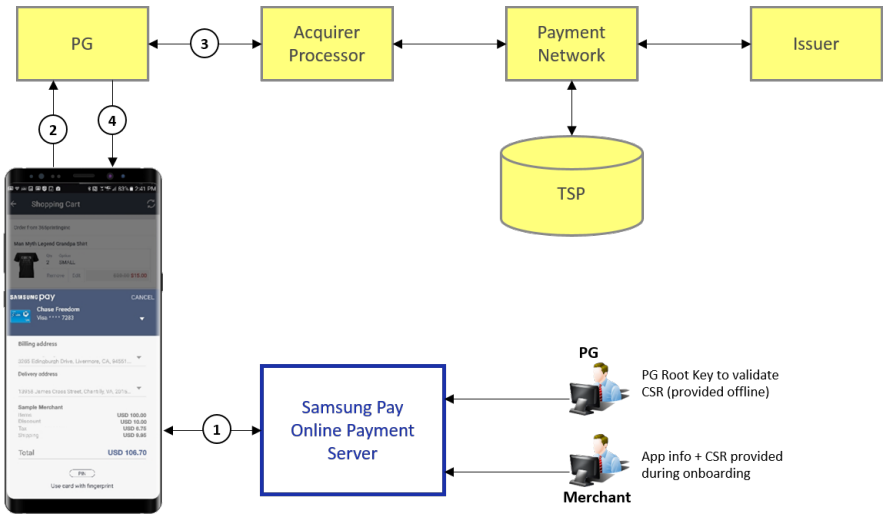

Figure 2a. Network token mode

Gateway Token mode (indirect)

1. In the merchant app at checkout, the user selects Samsung Pay as the payment method; the Samsung Pay app then requests partner verification from the Samsung Pay Online Payment Server.
2. Encrypted payment information and the Samsung Pay Service ID for the PG aggregator are passed to the Samsung-PG Interface Server.
3. The interface server sends a transaction authorization request to the PG on behalf of the merchant; the PG verifies the merchant ID before generating a transaction Reference ID.
4. The Reference ID is passed to the merchant app via an SDK callback; the merchant app then passes the Reference ID to the PG for payment processing execution.
5. The Samsung-PG Interface Server returns the payment token to the PG (this is the gateway token it received from the Samsung Pay app in Step 2).
6. The PG continues payment processing with the appropriate acquirer and payment network.
7. The PG passes the issuer's verdict (approve/decline) to the merchant app for display to the user.

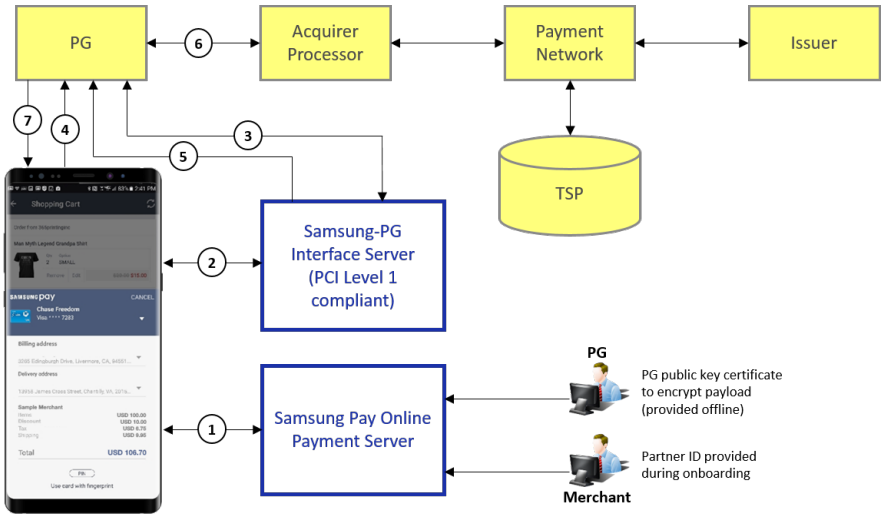

Figure 2b. Gateway token mode

Account preprovisioning

Account preprovisioning refers to secure onboarding of a new customer within the merchant app based on information provided by Samsung Pay. Using the In-App Payments service, the merchant app can leverage Samsung Pay API calls and callbacks to obtain a Samsung Pay user's profile information for new account creation, eliminating all or most manual entry. This service class can also request payment for the new customer's initial order (pending items in the user's shopping cart) with a payment card selected in Samsung Pay. In all cases, without the user's explicit authentication in Samsung Pay via secure PIN or biometric scan (fingerprint or iris), no personal information is shared with the merchant app and no payment is authorized.

PGs adopting the merchant aggregation model should consider embedding the Samsung Pay SDK mechanisms for account preprovisioning in the PG's SDK, along with the payment transaction elements, as a value-added service.

In any case, API calls and callbacks for account preprovisioning all take place locally on the device. No separate or special server interaction is necessary unless the user opts to make an initial purchase with Samsung Pay. When this is the case, Samsung Pay's tokenized payment logic is applied as discussed in the preceding section.

Communication workflow

Let's take a brief look at the end-to-end workflow between the merchant app and the PG Server using Samsung Pay's In-App Payments service; first for network token mode (Figure 3a) and then for gateway token mode (Figure 3b).

Direct mode - network token

Generated by the merchant app in direct mode and sent through the PG SDK, the payment request contains the transaction information necessary for PG payment processing — card brand (American Express, Discover, Mastercard, Visa), transaction amount, required addresses (billing/shipping), and so forth; the PG SDK then submits the required keys and other merchant information, as required, for merchant verification by the PG server. The result of merchant verification (success/failure) is returned to the PG SDK. It then sends

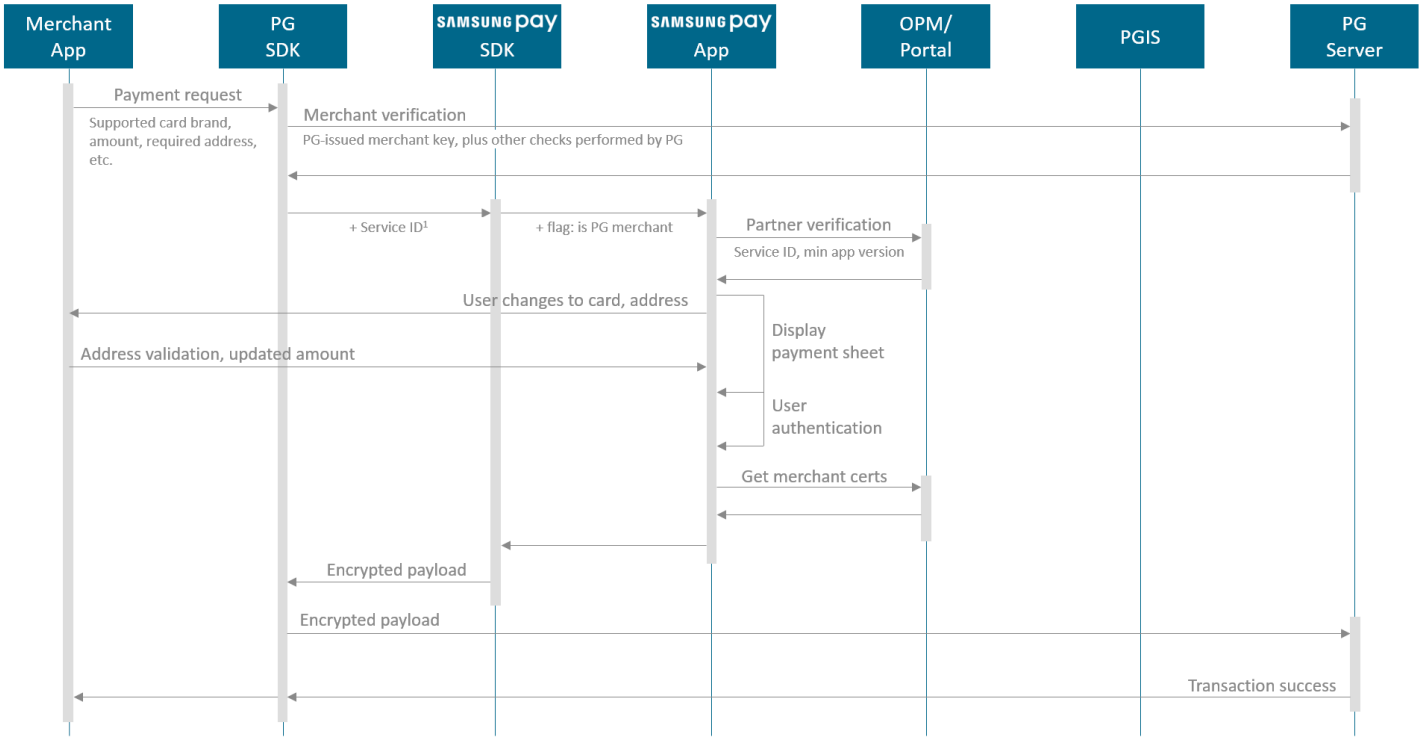

Figure 3a. Network token (direct) mode

the PG's **Service ID**[1] to the Samsung Pay SDK, which requests partner verification through the Samsung Pay app from the Samsung Pay Online Payment Manager (OPM)/Developers portal. Once the partner information is verified (**Service ID** matches supported PG SDK version), Samsung Pay displays the corresponding payment sheet on the mobile device for user authentication. If authentication is successful, the Samsung Pay app requests the merchant certificate on file with the OPM/Portal. Via the Samsung Pay SDK, the corresponding certificate is returned to the PG SDK, which sends the encrypted transaction payload to the PG Server, which processes payment through the designated payment network. Official bank/issuer approval is received from the PG Server in a callback to the PG SDK, which updates the merchant app for display to the user in a standard transaction success/failed message.

Indirect mode - gateway token

The flow for indirect mode is identical to the flow for direct mode up until the Samsung Pay app receives the merchant certs from the OPM/Portal. At that point, in contrast to direct mode, the Samsung Pay app sends its encrypted transaction payload to the PGIS, which authorizes its submission on behalf of the merchant; whereupon the PG Server generates a **Transaction Reference ID**, which is returned to the PG SDK through the Samsung Pay app and Samsung Pay SDK. The PG SDK initiates completion of the transaction by sending the PG-recognized **Merchant ID** and **Transaction Reference ID** to the PG Server for payment processing execution with the appropriate financial network. Once bank/issuer approval is received from the PG Server in a callback to the PG SDK, the merchant app updates its display with the appropriate transaction success/failed message.

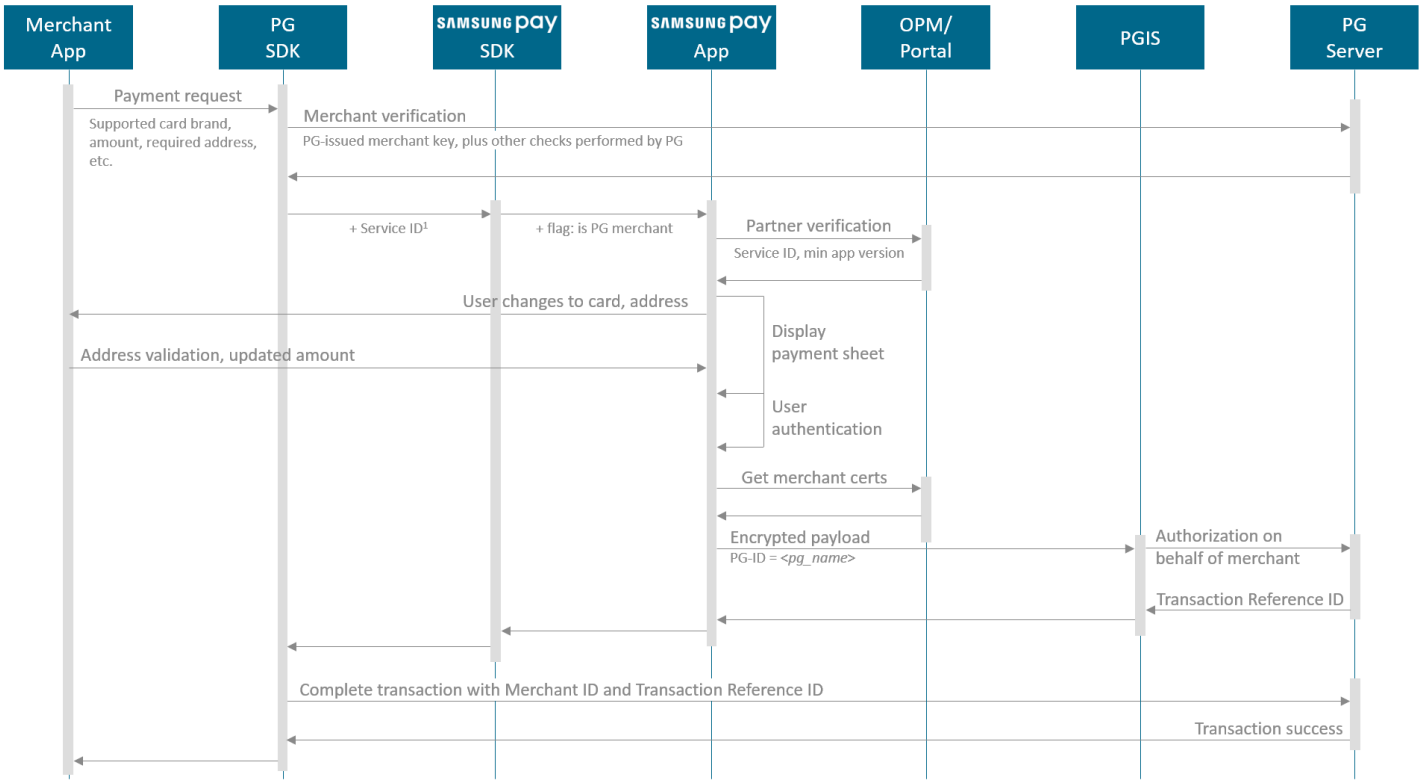

Figure 3b. Gateway token (indirect) mode

PGs can onboard with Samsung Pay under either the merchant aggregation or standalone (single) model — or both — although the aggregation model is strongly advocated for its ease of implementation and scalability. With aggregation, the PG sets up a single Samsung Pay **Service ID**, which it can then share with its participating merchants when they onboard with the PG. Meanwhile, the **Merchant ID** or registration code assigned to the merchant by the PG, along with any additional checks, verify the merchant and the merchant's mobile app to the PG's backend as a valid partner/vehicle for purposes of card payment processing.

1 For PGs using the aggregation model, there is one Samsung Pay **Service ID** for all requests, regardless of the merchant. If the PG is not aggregating merchants, or the merchant making the request is a standalone partner (i.e., not a participant in the PG's Samsung Pay aggregation program; onboarded separately with the Samsung Pay Developers portal), then the **Service ID** will be the one uniquely assigned to that particular merchant, which means the merchant must include it in the original payment request.

Next, let's take a look at the general use cases supported by the In-App Payments service. Please note that for the aggregation model, "merchant app" is presumed to be integrated with the PG SDK, which will embed or subsume the Samsung Pay SDK to work as described.

Use cases and recommended UX

Samsung Pay's In-App Payments service supports three fundamental use cases:

1. **Get Samsung Pay user information** to preprovision a new customer account in the merchant app.
2. **Get Samsung Pay user information AND request payment** for the new customer's initial order.
3. **Request standard in-app payment for a Samsung Pay user**; no user information needed for merchant account setup.

Case #3 typically applies to returning customers with established merchant accounts or new customers who opt to manually enter their user information for merchant account creation and then want to make a purchase. All three use cases assume that the user is already a Samsung Pay registered user and that the Samsung Pay app is installed on the same device running the merchant app.

Use Case #1: Get User Information

Story. As an online merchant, I want to import the customer profile information from Samsung Pay that I need to create a new account in my online store for a shopper who is also a Samsung Pay user, thereby sparing the shopper from having to manually enter data into my app that is already available from Samsung Pay. But, it's important that no user information be collected without the user's explicit consent via authentication using Samsung Pay's biometric scan or entering a valid Samsung Pay PIN. If authentication fails or times-out, the user information request must be aborted before any user data is transmitted to my merchant app.

Workflow. Figure 4 captures the essential workflow for account preprovisioning.

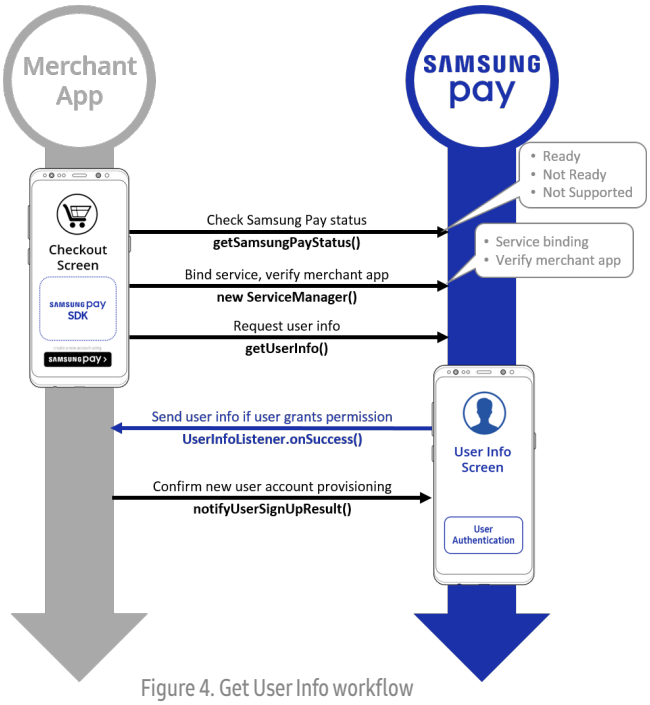

Figure 4. Get User Info workflow

First, via the PG's SDK, the merchant app checks the status of the Samsung Pay app by calling the **getSamsungPayStatus()** API, which returns one of the following values: **SPAY_READY**, **SPAY_NOT_READY**, or **SPAY_NOT_SUPPORTED**. A status of "ready" means the Samsung Pay app is installed on the device and setup is complete. "Not ready" means the app is installed but still needs to be set up by the user. "Not supported" means the device is not a Samsung device or is not an eligible Samsung device.

If Samsung Pay is "not ready" (in a stub-only state or not signed-in to a Samsung Account) or if Samsung Pay is not supported by the device, the merchant app should abort further preprovisioning steps[1]. If the status result is "ready," the merchant app can display a branded Samsung Pay button, presenting the user with the option of using Samsung Pay to create a new user account for the merchant's online store. If the user taps the Samsung Pay button, the merchant app calls the SDK's **PaymentManager** class (also used for transactions) to initiate a connection with the Samsung Pay app, binding the app to the In-App Payments service and requesting the user information in Samsung Pay with a call to **getUserInfo()**. On success, the listener callback returns a JSON string that the merchant app can parse into the constituent components of the user information the merchant app requested — name, billing address, shipping address, phone, email, or any combination thereof.

Recommended UX. Reflected in Figure 5 below, the user experience (UX) for this use case will need to be based on the merchant app's customer account sign-up/sign-in policy and procedure, and entails designating the merchants app's most appropriate UI entry point for account preprovisioning using Samsung Pay. A branded Samsung Pay button asset should only be displayed to users who don't yet have an account with your store but are verified Samsung Pay users. The latter is confirmed with a **getSamsungPayStatus()** API call that checks whether a supported version of the Samsung Pay app is installed on the device and in "ready" status.

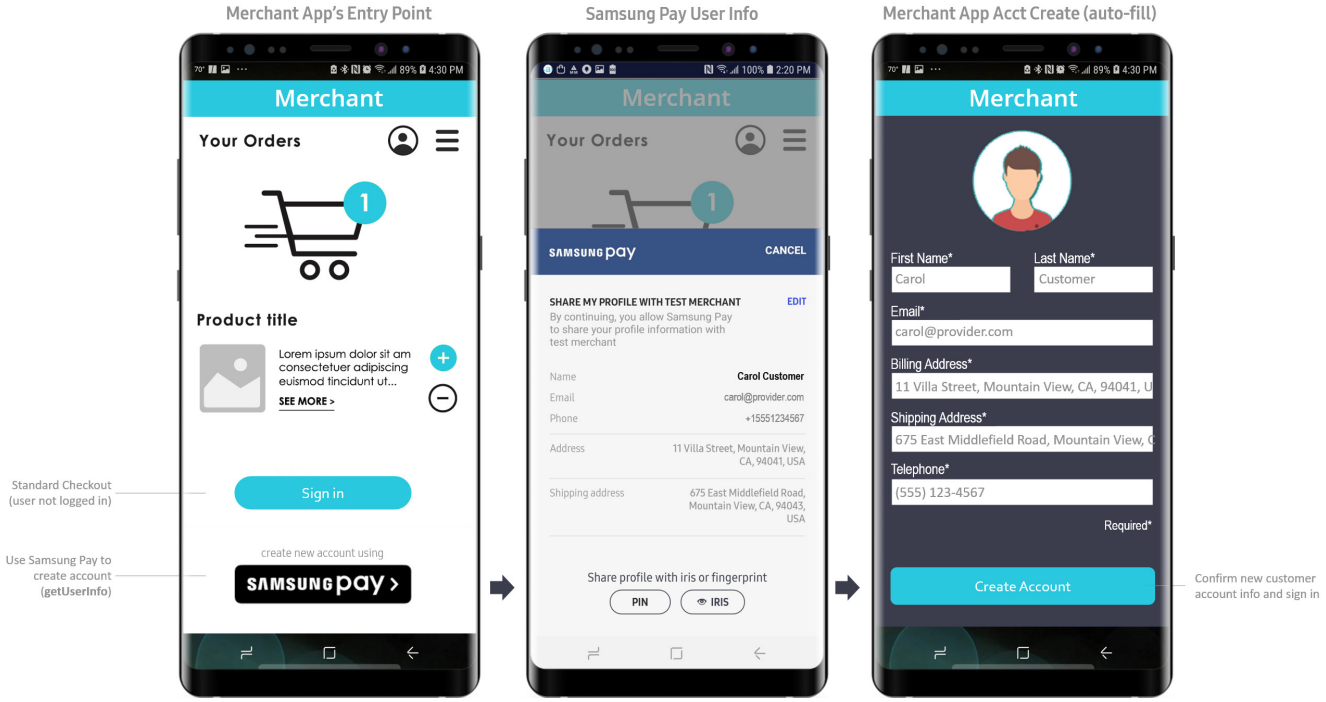

Figure 5. Get User Info UX

If the Samsung Pay app is "ready," the merchant app should then display a branded "create new account using Samsung Pay" button that presents the user with the option of creating/registering a new online store account using information in Samsung Pay. If the user taps the option, the merchant app makes the necessary API calls to have Samsung Pay display a "User Info" screen, which allows the user to edit the information before granting permission to share the data by authenticating the action with a fingerprint/iris scan or entering a valid Samsung Pay PIN.

1 Although not included in the guidance presented here, when the status of the Samsung Pay app is **SPAY_NOT_READY**, the PG's SDK can support the option of calling either the **activateSamsungPay()** method to launch the pay app so the user can register their Samsung Account and enroll at least one payment card, or **updateSamsungPay()** to let the user upgrade to a version of the pay app that is compatible with the respective versions of the merchant app and the Samsung Pay SDK it implements. Refer to the SDK's Javadoc reference and *Samsung Pay SDK Programming Guide* for additional details and guidance.

Use Case #2: Get User Information AND request payment

Initiating a payment request after account preprovisioning offers even more user convenience.

Story. As an online merchant that supports tokenized payments, I want to request payment on behalf of a new shopper for items in my online store using Samsung Pay. At the same time and with the user's explicit permission, I want to import the user's profile information already on file in Samsung Pay in order to create a new customer account in my online store so the user isn't required to manually enter the information into my account creation form.

Workflow. As with the previous use case, when the Samsung Pay app is in "ready" status on the device, the merchant app displays a branded Samsung Pay button at an appropriate UI entry point. In contrast to Use Case #1, however, the button here should indicate *both* account creation and checkout using Samsung Pay in accordance with Samsung Pay branding guidelines. Use Case #3, discussed later, only needs to indicate the payment method option.

Otherwise, the payment request for pending items in the user's shopping cart is initiated by calling **startInAppPayWithUserInfo()** with a **customSheetPaymentInfo** object, **RequestType.PAYMENT_CARD** and/or **RequestType.NO_SHEET_UPDATE_CALLBACK**, and a **userInfoListener** callback. Samsung Pay will then generate the transaction payload along with the user information request for account preprovisioning.

Presuming the user information is available and the request is successfully received, both payment and user information is displayed for card selection and/or editing of the user's profile information. User authentication with PIN or biometric scan transmits the user information to the merchant app together with the user's transaction authorization. Although the SDK's **CustomSheet** object, sent via a **startInAppPayWithUserInfo()** call supports address and amount controls, we recommend display of the transaction total amount only.

Other **SheetItemTypes** for standard In-App Payments discussed in Use Case #3 — specifically, the controls for a spinner (subscription/recurring payments) and plain text content — are not supported for account preprovisioning. The Fast Checkout (FCO) feature is also not supported in a **startInAppPayWithUserInfo()** call.

As with the **getUserInfo()** method discussed in Use Case #1, in a **startInAppPayWithUserInfo()** call the user information for new account creation is returned in the **userInfoListener** callback. The merchant app then calls **notifyUserSignUpResult()** to transmit the user's account creation result back to Samsung Pay.

For the payment component, the transaction result is included in **transactionInfoListener**. If the PG supports network tokens (direct mode), the transaction payload object returned in **transactionInfoListener.onSuccess()** will include encrypted payment credentials. If the PG supports gateway tokens, the payload includes the PG's token reference ID for payment processing. These are then submitted to the PG's backend via its proprietary SDK using the direct or indirect mode, respectively, described above under Tokenized payments.

Once the payment request has been processed, the PG sends back the result (transaction success/failure) directly through its PG SDK as described and illustrated in the communication workflow for the respective direct/indirect mode supported.

Figure 6 shows the general workflow for Use Case #2. Again, as reflected on the next page, it's important to remember that the callback for obtaining payment information (**transactionInfoListener**) is distinct from the callback for obtaining user information (**userInfoListener**).

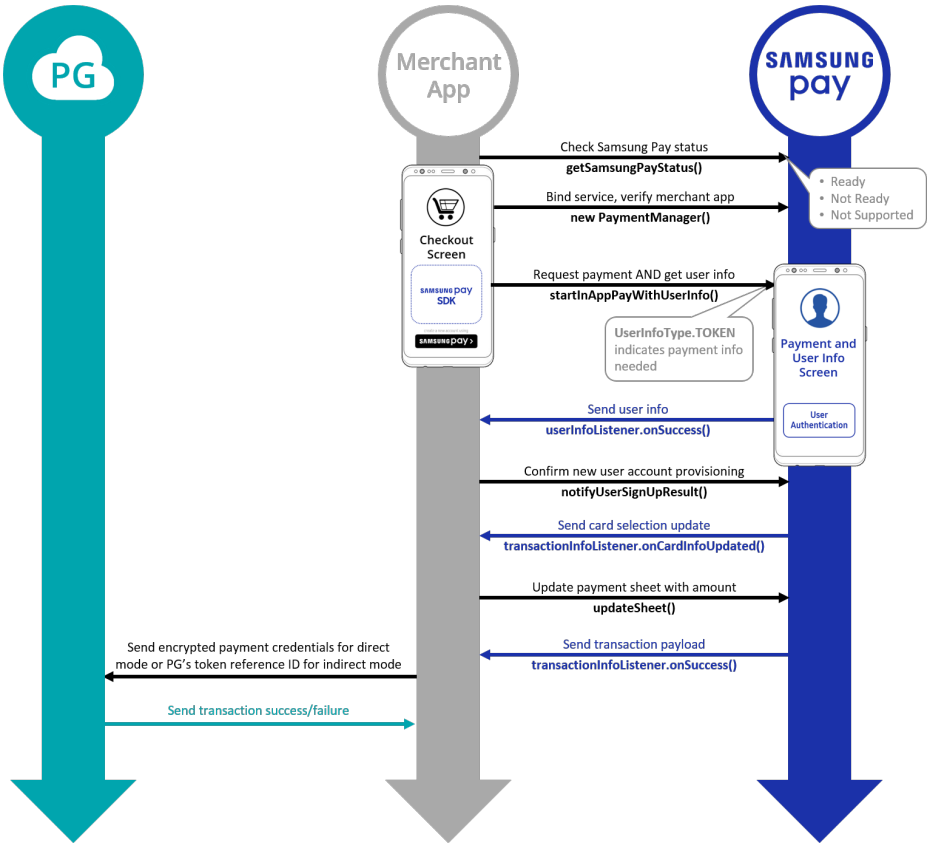

Figure 6. Get User Info AND Request Payment workflow

Recommended UX. Reflected in Figure 7, the suggested UX directly parallels Use Case #1 but is extended to present an abbreviated payment sheet for secure user authentication. Any user changes to the selected card, billing and/or shipping addresses made in Samsung Pay are contained in **cardInfoUpdated()** callbacks. If these changes result in an adjusted transaction amount — for instance, if the merchant offers a discount for a credit card brand like Visa® as opposed to other card brands, or if a different shipping address incurs a change in the shipping charge (+/-) — the merchant app calls the SDK's **updateSheet()** method to send Samsung Pay the adjusted transaction amount.

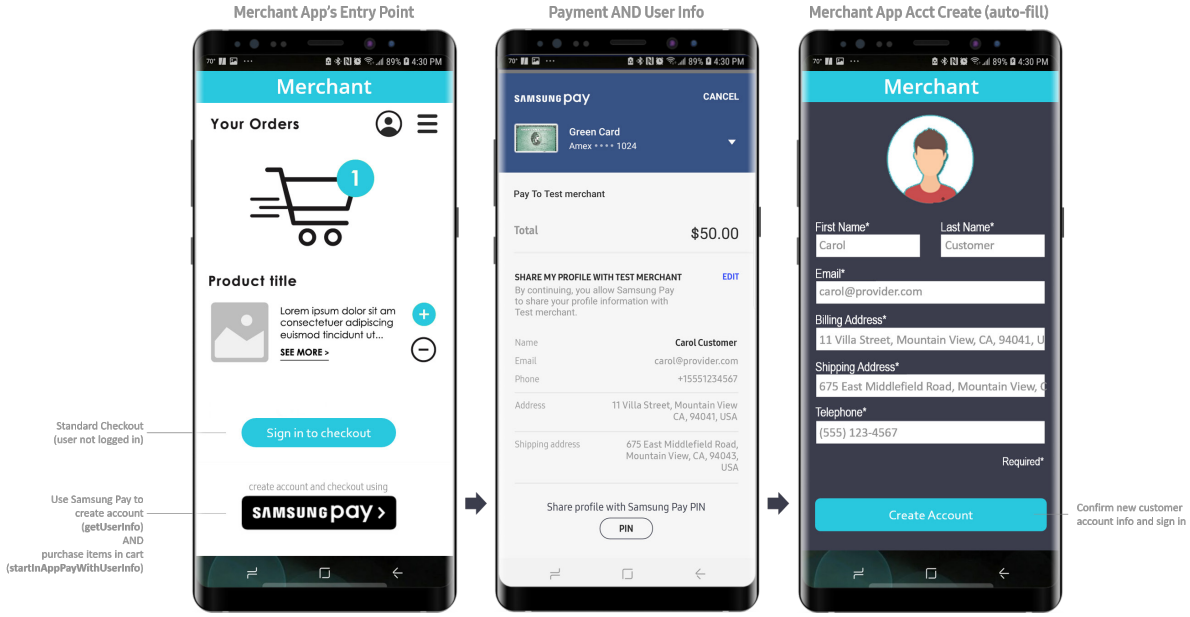

Figure 7. Get User Info AND Request Payment UX

Use Case #3: Request standard in-app payment
(no user information needed for account preprovisioning)

This use case provides Samsung Pay as a standard payment method when there's no need for preprovisioning information (i.e., the user already has an account with the merchant's online store). Here, a payment request can be made as soon as the merchant app checks Samsung Pay app status on the device and determines that it is in a "ready" state.

Story. As an online merchant supporting tokenized payments, I want to request payment using Samsung Pay on behalf of a returning shopper for any items in the user's shopping cart selected for checkout and payment.

Workflow. For this use case, you'll need to take the following steps:

1. Check the ready status of the Samsung Pay app on the device.
2. Start the Payment Manager to establish the service binding and verify the merchant app.
3. Get payment card information and the transaction amount, including updates.
4. Get/update the user's shipping address if shipping charges are incurred (presumes flat-rate shipping is not offered).
5. Authenticate the user.
6. Submit payment information to your PG.
7. Verify transaction success or failure.

The general workflow for this use case is shown in Figure 8.

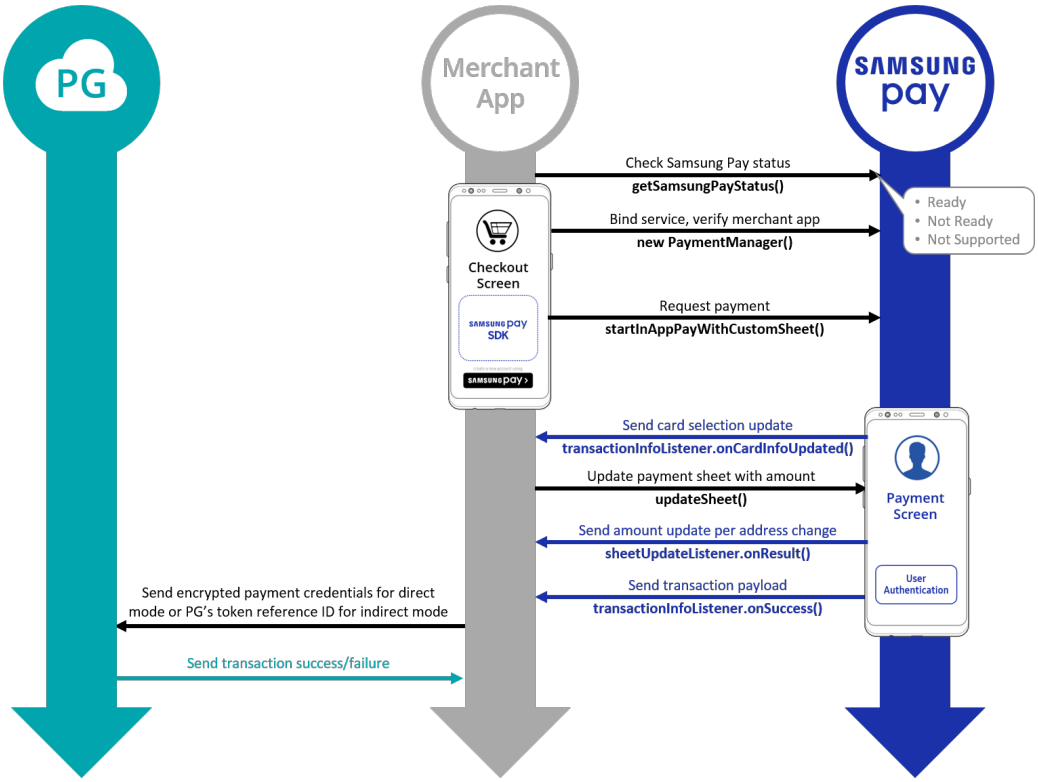

Figure 8. Request payment for a Samsung Pay user workflow

For this use case, the callbacks are implemented using the **CustomSheetTransactionListener** method, rather than the abridged **TransactionInfoListener** for a new customer's initial order implemented for Use Case #2.

Recommended UX. Figure 9 shows the suggested UX for payment requests in which no user information is needed. Card selection and changes to the **Billing address** and **Delivery address** are supported in the payment sheet UI. However, callback iterations will need to be handled by the merchant app when the user makes repeated changes to the information on the payment sheet requiring transaction amount updates prior to user authentication.

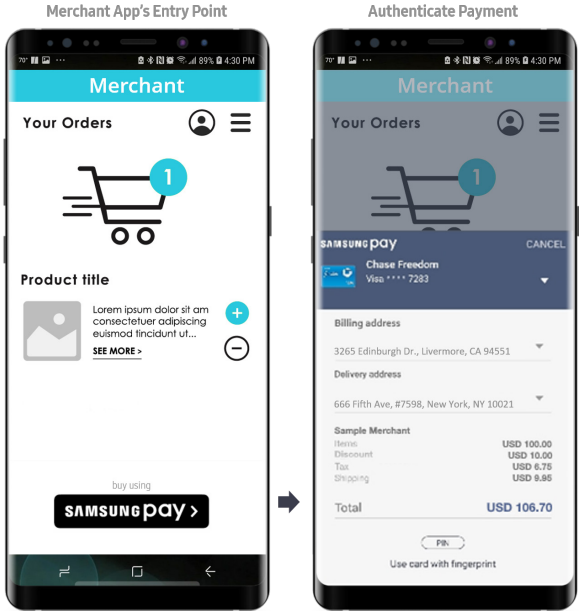

Figure 9. Standard In-App Payment request UX — no account preprovisioning required

Keeping these three primary use cases in mind, we can now delve into SDK-within-SDK integration for merchant aggregation.

Samsung Pay token data

The data included in the token provided to the PG by Samsung Pay, includes the following fields:

Field	Definition/Description	Required (Y/N)	Sample Value
merchantRef	Merchant reference value	Y	*<pg_assigned_value>*
cardNumber	DPAN	Y	4111111111111111
exp	Expiration date	Y	0420
cryptogram	Samsung Pay cryptogram	Y	AK+zkbPMCORcABCD3AGRAoACFA==
eci	3D-secure electronic commerce indicator (ECI)	Y	5
tokenizationMethod	Method used for generating token	Y	samsung_pay
utc	Coordinated universal time (UTC) - timestamp	N	2147483647
merchantName	Name of originating merchant	N	David's Burgers
appPackageName	Application package name (merchant app)	N	com.davidsburgers.orderahead
appSignature	Application signature (signed by developer)	N	cert.rsa of application
merchantDomain	Merchant domain	N	checkout.davidsburgers.com

Summary of integration and processing tasks/roles

A successful in-app integration requires participation and critical contributions from each of the partners involved — merchant, PG, and Samsung Pay. For the merchant aggregation model, the PG does much of the heavy lifting with its SDK to correctly configure the merchant's transaction request for payment using Samsung Pay, standardizing the integration process and thereby saving its aggregated merchants significant development time.

The following table summarizes the principal tasks and contributions needed for a successful outcome.

Phase	Merchant	Merchant Aggregator PG	Samsung Pay
Onboarding	• Enable Samsung Pay in PG portal • Update Android project with PG's newest SDK for Android • Add Samsung Pay tag to project's Android manifest	• Provide website/portal for merchant registration and account administration	• Provide a common **serviceId** to PG for all of its aggregated merchants
Initialization	• Initialize PG SDK using tokenization key	• Fetch In-App **serviceId** (not hard-coded or cached) • Check if merchant app is blacklisted	
Check readiness	• Call PG method (e.g., **SamsungPay#isReadyToPay**) • Pass in-test versus release flag	• Retrieve merchant-supported card brands and pass to Samsung Pay SDK • Use **serviceId** to check Samsung Pay status • Use merchant-supported card brands to match user-enrolled card brands in Samsung Pay • Check Fast Checkout (FCO) status • Configure PG SDK to display Samsung Pay button or Samsung Pay FCO button, plus last 4 digits of card	• Validate PG **serviceId** and relax merchant app verification checks
Create transaction	• Call PG method (e.g., **SamsungPay#requestPayment**) • Pass-in transaction amount • Pass-in merchant display name • Pass-in shipping address required flag (optional; default = **False**) • Pass-in FCO enabled flag (optional; default = **False**) • Pass-in contact info required flag (optional; default = **False**)	• Register **TransactionInfoListener** • Call **startInAppPayWithCustomSheet** with merchant-supplied parameters • Set **AddressInPaymentSheet** option to **NEED_SHIPPING_SPAY** if flag = **True**; otherwise, default = **NEED_BILLING_SPAY** (see note below) • Receive **paymentCredential** from Samsung Pay, including nonce • Retrieve other info (email, phone, shipping address) from Samsung Pay based on merchant request	• Check test vs. release setting and connect with appropriate PG environment (test or production) • Send token data • Return PG nonce within **paymentCredential**
Create account (Use Case #1 only)	• Call PG method (e.g., SamsungPay#requestAccount) • Pass-in shipping address required flag (optional; default = **False**) • Pass-in FCO enabled flag (optional; default = **False**) • Pass-in payment not required flag (optional; default = **False**)	• Register **AccountInfoListener** • Call **startInAppPayWithUserInfo** with merchant-supplied parameters • Set **AddressInPaymentSheet** option to **NEED_SHIPPING_SPAY** if flag = **True**; otherwise, default = **NEED_BILLING_SPAY** (see note below) • Receive **paymentCredential** from Samsung Pay, including nonce, unless excluded • Retrieve other info (email, phone, shipping address) from Samsung Pay based on merchant request	• Check test vs. release setting and connect with appropriate PG environment (test or production) • Send token data • Return PG nonce within **paymentCredential**

Note: When the shipping address is provided by the merchant app (shipping address required flag = **False**; **AddressInPaymentSheet** option is set to **SEND_SHIPPING** or **NEED_BILLING_SEND_SHIPPING**), it is not user-editable in the payment sheet. Ergo, the shipping fee (if applicable) must be pre-calculated by the merchant app and included in the total amount.

Integration steps for merchant aggregation

Onboarding a merchant aggregation PG is a joint activity involving the engineering and business teams of both the PG and Samsung Pay. Because a one-size-fits-all implementation is impractical and standardization is nascent, the integration effort must be collaborative.

That said, there are some general "rules of thumb" and common sense best practices.

- For the merchant app developer, onboarding with an aggregator PG should be no more complicated than adding two external libraries (**.jar** files) — the PG's SDK and the Samsung Pay SDK — to the merchant's existing Android project, declaring the dependencies, and inserting (coding) the necessary PG API calls/callbacks needed to support Samsung Pay as an online payment method.

  Note: Aggregated merchants do not sign-up or onboard with Samsung Pay directly; configuration must be handled internally by the PG's SDK using the Samsung Pay SDK as an external third-party library.

- As an aggregator PG, you'll need to support both a sandbox and a production environment, typically enabled from a dashboard or settings menu controlled from the PG's backend. In the sandbox environment, Samsung Pay will return valid testing nonces that point to dummy account data so that the card selected in Samsung Pay is not charged.

- From its onboarding website, the aggregator PG should provide access to Samsung Pay's branding guidelines and button assets.

  Note: The use of fragments or sub-activities to manage Samsung Pay interactions is highly recommended. Please also note that the Samsung Pay SDK is designed exclusively for Samsung mobile devices supporting Samsung Pay and running Android Nougat 7.1.2 (API level 24–25) or later versions of the Android OS. A comprehensive Javadoc API reference is included with the Samsung Pay SDK.

Step 1. Configure your project to import the Samsung Pay SDK

A. Configure Android Studio to include the Samsung Pay SDK as an external library with either a local dependency or a remote dependency.

To add it as a local dependency:

- Download the Samsung Pay SDK from https://pay.samsung.com/developers and extract its contents to the desired destination directory on your local machine.
- Copy **samsungpay.jar** from the **libs** folder of the destination directory to the **libs** folder of your Android project.

To add it as a remote dependency:

Open your project-level **build.gradle** file and declare the repository provided by your Samsung Pay representative.

```
repositories {
    <repository> {<url 'url_address_string'>} // e.g., maven {url 'https://maven.spaysdk.io/public'}
}
```

B. In **build.gradle**, add the following dependency:

```
dependencies
     compile files("libs/samsungpay.jar")
{
```

Next, sync your Gradle files, either by clicking the **Sync** banner or by selecting the **Sync Project with Gradle Files** icon in the toolbar.

C. Import the Samsung Pay SDK into your code with:

```
import com.samsung.android.sdk.samsungpay.v2;
```

D. Next, modify the metadata in your project's manifest file[1].

```
<application>
     <meta-data
          android:name="debug_mode" // used for standalone merchant app testing
          android:value="N" />      // not needed for merchant aggregation testing
     <meta-data
          android:name="spay_sdk_api_level"
          android:value="2.6" />    //  max api_level -- very important
</application>
```

Here, **API Level** ("spay_sdk_api_level") governs the API dependency based on **Country** and **Service Type** to optimize version control and to improve backward compatibility. This means that, upon integrating with the latest Samsung Pay SDK update, your PG SDK can continue to use APIs based in previous levels. However, the API level specified in the manifest is the forward limit on compatibility. In other words, setting the API level at 2.5 provides SDK support for all earlier versions (currently down to 1.4) but will block access to APIs in level 2.6. Consequently, if you want to take advantage of the SDK's most recently added APIs, you'll need to change the API level accordingly in the manifest file and do a new build (recompile) of your JAR.

Important: When you deploy, be sure to notify your aggregated merchants so they can upgrade their apps to your new SDK.

Step 2. Create the Samsung Pay instance and check app status

A. Set the **serviceId** (provided by your Samsung Pay rep) and **ServiceType** (INAPP_PAYMENT) in **PartnerInfo** before calling any other APIs. Do not confuse **serviceId** with **AppId**; the two are mutually exclusive. This is an implicit call made by the PG's SDK on behalf of the merchant app as shown in the following example (your merchant aggregator PG **serviceId** will be unique).

```
String serviceID = "5c104c52bb944d8bb33be4"  // aggregator PG's Samsung Pay service ID
Bundle bundle = new Bundle();
bundle.putString(SamsungPay.PARTNER_SERVICE_TYPE, SpaySdk.ServiceType.INAPP_PAYMENT.toString());
bundle.putString(SpaySdk.EXTRA_PARTNER_NAME, "Your Merchant Name");
PartnerInfo partnerInfo = new PartnerInfo(serviceId, bundle);
```

B. Check Samsung Pay app status on the device by calling the **getSamsungPayStatus()** method of the **SamsungPay** class. This is another implicit call made on behalf of the merchant app that must be done before calling any other feature of the Samsung Pay SDK.

```
void getSamsungPayStatus(StatusListener callback)
```

Determine if the **onSuccess()** value of **StatusListener** is **SPAY_READY**, **SPAY_NOT_READY**, or **SPAY_NOT_SUPPORTED** and take appropriate action. Use the following example to structure the call.

```
samsungPay.getSamsungPayStatus(new StatusListener() {
   @Override
   public void onSuccess(int status, Bundle bundle) {
      switch (status) {
         case SamsungPay.SPAY_NOT_SUPPORTED:
            // Samsung Pay is not supported by the device--do not show Samsung Pay button
            break;
```

1 Be sure to place **<meta-data>** tags inside the **<application>** tag.

```
            case SamsungPay.SPAY_NOT_READY:
               // Samsung Pay not activated on the device or app version not updated
               // Either activate Pay app or update it--see Javadoc for both methods--
               // otherwise, don't show Samsung Pay button
               break;
            case SamsungPay.SPAY_READY:
               // Samsung Pay is ready--show button
               break;
            default:
               // Unexpected result--do not show button
               break;
         }
      }
   @Override
      public void onFail(int errorCode, Bundle bundle) {
         samsungPayButton.setVisibility(View.INVISIBLE);
         Log.d(TAG, "checkSamsungPayStatus onFail() : " + errorCode);
      }
   });
```

Step 3. Support Use Case #1 - Get user profile information

To obtain the user profile information needed to auto-fill the merchant app's account form without requesting payment from Samsung Pay, call **getUserInfo()**.

```
public void getUserInfo (RequestType[] requiredTypes, UserInfoListener listener)
```

When this API is called, the Samsung Pay app displays the requested information (Figure 10). If any information is absent, users can manually enter the missing fields required to establish their account by tapping **EDIT**. Authenticating with a Samsung Pay PIN or other biometric method (fingerprint, iris) confirms the user's permission to share the information and transmit it to your partner app.

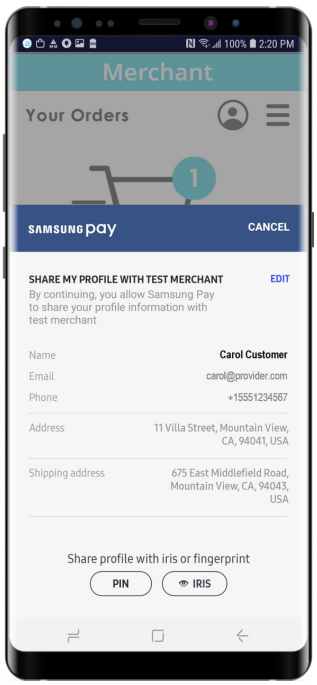

Figure 10. Samsung Pay User Info screen

RequestType[] can specify **ALL** or any combination of the following fields (listed here alphabetically):

- **BILLING_ADDRESS** – current billing address registered with the user's Samsung Account
- **EMAIL** – email address registered with the user's Samsung Account
- **NAME** – first and last name registered with the user's Samsung Account
- **SHIPPING_ADDRESS** – current or last selected shipping address on file in Samsung Pay
- **TEL** – telephone number registered with the user's Samsung Account.

Below is an example of a call using the **getUserInfo()** method of the **ServiceManager** class that requests the user's information by item, instead of using the **ALL** option. The requested information is received in a corresponding **UserInfoListener()** callback. Use the following example to aid you with the structure of the API call.

```
UserInfoListener userInfoListener = new UserInfoListener() {
   @Override
   public void onSuccess(UserInfoCollection userInfo) {
      for (RequestType type : userInfo.getAvailableTypes()) {
          switch (type) {
```

```
                case: DATE_OF_BIRTH:
                   // userInfo.getDate(type)
                   break;
                case BILLING_ADDRESS:
                case SHIPPING_ADDRESS
                   // Address address = userInfo.getAddress(type);
                   break;
                default:
                  // userInfo.getString(type)
                   break;
             }
      }
      mServiceManager.notifyUserSignUpResult("", true, new UserSignUpNotifyListener() {
         @Override
         public void onSuccess() {
            Log.d(TAG, "notifyUserSignUpResult onSuccess ");
         }

         @Override
         public void onFail(int i, Bundle bundle) {
            Log.d(TAG, "notifyUserSignUpResult onFail "+i);
         }
      });

      @Override
         public void onFail(final int i, Bundle bundle) {
            // show failure msg
            mServiceManager.notifyUserSignUpResult("", false, new UserSignUpNotifyListener() {
               @Override
               public void onSuccess() {
                  Log.d(TAG, "notifyUserSignUpResult onSuccess ");
               }

               @Override
               public void onFail(int i, Bundle bundle) {
                  Log.d(TAG, "notifyUserSignUpResult onFail "+i);
               }
             });
          }
      };

private void startSamsungPayForSignupInfo() {

// Check Samsung Pay status here; if SPAY_READY, show branded button, then do the following:

      RequestType[] requestTypes = null;
      requestTypes = new RequestType[]{RequestType.NAME, RequestType.EMAIL,
         requestType.BILLING_ADDRESS, RequestType.SHIPPING_ADDRESS, RequestType.TEL};
      // << or >>
      // requestTypes = new RequestType[]{RequestType.ALL};
      // create new Samsung Pay instance here
```

```
        mServiceManager = new ServiceManager(this, partnerInfo);
        mServiceManager.getUserInfo(requestTypes, userInfoListener);
    }
```

The merchant app can parse the data received in **userInfoListener()** to auto-fill its new account form for optional user verification and submission. For the example UX, see Figure 5.

Step 4. Support Use Case #2 – Get user info AND request payment

To support merchant requests for user account preprovisioning, as well as to request payment for the user's initial order (current selections in the user's shopping cart), call the **startInAppPayWithUserInfo()** method of the **PaymentManager** class. This sends a request for both the payment credential (equivalent to **startInAppPayWithCustomSheet()** implemented for Use Case #3) and the user's personal information (same as **getUserInfo**) based on a single authentication confirmation implemented for Use Case #1.

Here, your partner app has the option of receiving many of the same callbacks included with **startInAppPayWithCustomSheet()**. Those familiar with standard in-app payments for non aggregated merchants should be aware that, in the US market, the **CustomSheet** object sent in the **startInAppPayWithUserInfo()** API call only supports an **AddressControl** and an **AmountBoxControl**. The other **SheetItemTypes** are not supported, and are therefore not displayed in the payment sheet. The Fast Checkout (FCO) feature is also not supported in a **startInAppPayWithUserInfo()** call. Given that the user profile information is also displayed in the UI, it is strongly recommended that only the total transaction amount is shown for this use case, rather than a complete price itemization supported by Use Case #3.

Otherwise, the available **RequestType**s you can use here are the same as those listed above for Use Case #1, although for payment requests, the following additional fields are added:

- **NO_SHEET_UPDATE_CALLBACK** – turns off callbacks when payment information is updated by the user in Samsung Pay; i.e., card or address changes)
- **PAYMENT_CARD** – returns payment card information for the card selected in Samsung Pay. Please note that this field has no effect when included in **getUserInfo()** — i.e., for Use Case #1 — so you can combine calls from your PG SDK for both use cases (#1 and #2) using this field without any adverse behavior.

Structure the code for implementing the **startInAppPayWithUserInfo()** method and build payment sheet support using the following sample code as your template.

```
private PaymentManager.CustomSheetTransactionInfoListener transactionListener
          = new PaymentManager.CustomSheetTransactionInfoListener() {
    // Callback received when the user changes the card on the payment sheet in Samsung Pay
    @Override
    public void onCardInfoUpdated(CardInfo selectedCardInfo, CustomSheet customSheet) {
       // Called when the user changes cards in Samsung Pay.
       // Newly selected cardInfo is passed so merchant app can update transaction amount
       // based on the different card (e.g., special discount, free shipping), as appropriate.
    }
    // This next callback is received when the payment is approved/authenticated by the user and
    // the transaction payload is generated. Payload can be an encrypted cryptogram (network token
    // mode) or the PG's token reference ID (gateway token mode).
    @Override
    public void onSuccess(CustomSheetPaymentInfo response, String paymentCredential,
          Bundle extraPaymentData) {
       // Called when Samsung Pay creates the transaction cryptogram, which merchant app
       // then sends to merchant server or PG to complete in-app payment.
    }
```

```
    @Override
    public void onFailure(int errorCode, Bundle errorData) {
        // Called when an error occurs during cryptogram generation
    }
};
UserInfoListener userInfoListener = new UserInfoListener() {
    @Override
    public void onSuccess(UserInfoCollection userInfo) {
        for (RequestType type : userInfo.getAvailableTypes()) {
            switch (type) {
                case: DATE_OF_BIRTH:
                    // userInfo.getDate(type)
                    break;
                case BILLING_ADDRESS:
                case SHIPPING_ADDRESS
                    // Address address = userInfo.getAddress(type);
                    break;
                default:
                    // userInfo.getString(type)
                break;
            }
        }

    @Override
    public void onFail(int errorCode, Bundle errorExtra) {
        Log.e(TAG, "onFail callback is called, errorCode: " + errorCode);
        // To get more reasons for the failure,
        // check the extra error codes in the errorData bundle, such as
        // SamsungPay.EXTRA_ERROR_REASON
    }
};

private void startSamsungPayForSignupInfo() {

    RequestType[] requestTypes = null;
         requestTypes = new RequestType[]{RequestType.NAME, RequestType.EMAIL,
            RequestType.BILLING_ADDRESS, RequestType.SHIPPING_ADDRESS,
            RequestType.TEL, RequestType.NO_SHEET_UPDATE_CALLBACK, RequestType.PAYMENT_CARD};
         // <<<<<<<<<<  or >>>>>>>>>
         // requestTypes = new RequestType[]{RequestType.ALL};

    // create new Samsung Pay instance here (see Step 2)

    mPaymentManager.startInAppWithUserInfo(paymentInfo.build(), transactionInfoListener,
        requestTypes, userInfoListener);
}
```

As with the **getUserInfo()** method, in a **startInAppPayWithUserInfo()** call, the user information for new account creation is returned in a **userInfoListener** callback. For the payment component, the transaction result is delivered to **transactionListener**, which provides the following events:

- **onCardInfoUpdated()** – called when the user changes the payment card. In this callback, the **updateSheet()** method must be called to update the current payment sheet in cases where the selected card and/or address is changed by the user.
- **onSuccess()** – called when Samsung Pay confirms payment. It provides the **CustomSheetPaymentInfo** object and the **paymentCredential** JSON string (see sample payment credential).

  CustomSheetPaymentInfo (Figure 11) represents the current transaction only. Although only a total transaction amount is recommended for display, **CustomSheetPaymentInfo** returns **amount**, **shippingAddress**, **merchantId**, **merchantName**, and **orderNumber**. Additional API methods available in the **onSuccess()** callback are:

  – **getPaymentCardLast4DPAN()** – returns the last 4 digits of the digitized personal/primary identification number (DPAN)
  – **getPaymentCardLast4FPAN()** – returns the last 4 digits of the funding personal/primary identification number (FPAN)
  – **getPaymentCardBrand()** – returns the brand of the card used for the transaction
  – **getPaymentCurrencyCode()** – returns the ISO currency code in which the transaction is valued
  – **getPaymentShippingAddress()** – returns the shipping/delivery address for the transaction
  – **getPaymentShippingMethod()** – returns the shipping method for the transaction.

  For PGs employing the direct model (network tokens), the **paymentCredential** is a JSON object containing an encrypted cryptogram which the merchant app should pass back to the PG. PGs using the indirect model (gateway tokens) rely on a JSON object containing a **reference ID** (card reference – a token ID generated by the PG) and status (i.e., **AUTHORIZED**, **PENDING**, **CHARGED**, or **REFUNDED**).

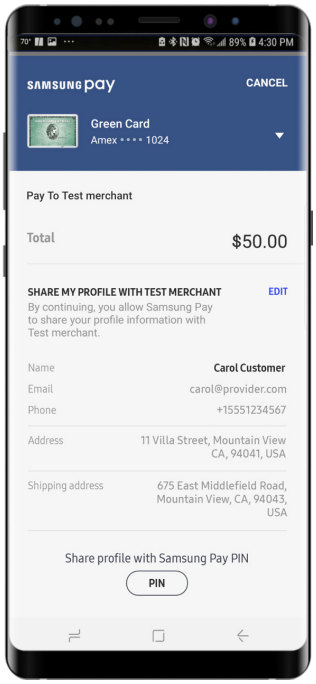

Figure 11. Payment Info AND User Info

- **onFailure()** – called when the transaction fails; returns the error code and **errorData** bundle for the failure.

Remember to consult the latest Javadoc SDK reference for complete class, object and method definitions with additional sample code related to all three use cases discussed in this document.

Step 5. Support Use Case #3: Request standard in-app payment

This use case does not require/request user profile information. It assumes a merchant store account for the user is already provisioned and the user is ready to checkout upon selecting Samsung Pay as the payment method. For this case, there are two primary tasks:

1. Build the payment sheet
2. Create the transaction request

In this case, the aggregator PG can opt to let the merchant app explicitly configure the payment sheet for Samsung Pay after adding **samsungpay.jar** as an external library in the merchant app's own IDE.

Building the payment sheet. The payment sheet is governed by the **CustomSheetPaymentInfo** class, which includes an **AddressControl** for billing and shipping addresses; an **AmountBoxControl** for listing multiple line items, subtotals, and a transaction grand total; a **PlainTextControl** for custom messaging; and a **SpinnerControl** for selecting different shipping methods. (See Configuring payment sheet controls below for more on each sheet control. See Handling user-entered changes for guidance on how the payment sheet is updated based on user inputs to the payment sheet controls.)

Use the following example to aid you in structuring the desired mechanisms within your PG SDK.

```
// Create payment sheet controls and content
private CustomSheetPaymentInfo makeCustomSheetPaymentInfo() {

   ArrayList<PaymentManager.Brand> bList = new ArrayList<>();
   // Specify the card brands supported; if no brands are specified, all brands are allowed
   brandList.add(PaymentManager.Brand.VISA);
   brandList.add(PaymentManager.Brand.MASTERCARD);
   brandList.add(PaymentManager.Brand.AMERICANEXPRESS);
   // Next, build the sheet controls in sequence; AmountBoxControl must be last
   CustomSheet customSheet = new CustomSheet();
   customSheet.addControl(makeBillingAddressControl());
   customSheet.addControl(makeShippingAddressControl());
   customSheet.addControl(makePlainTextControl());
   customSheet.addControl(makeShippingMethodSpinnerControl());
   customSheet.addControl(makeAmountControl()); // must be last

   // Finally, build the payment sheet
   CustomSheetPaymentInfo customSheetPaymentInfo = new CustomSheetPaymentInfo.Builder()
      .setMerchantId("123456")
      .setMerchantName("Sample Merchant")
      .setOrderNumber("AMZ007MAR")
      .setPaymentProtocol(CustomSheetPaymentInfo.PaymentProtocol.PROTOCOL_3DS)
      // Show both billing and shipping address
      .setAddressInPaymentSheet(CustomSheetPaymentInfo.AddressInPaymentSheet.NEED_BILLING_AND_SHIPPING)
      .setAllowedCardBrands(bList)
      .setCardHolderNameEnabled(true)
      .setRecurringEnabled(false)
      .setCustomSheet(customSheet)
      .build();
   return customSheetPaymentInfo;
}
```

Next, the aggregator PG's SDK will need to support the mechanisms needed to create the transaction request.

Creating the transaction request. Request a transaction using the **startInAppPayWithCustomSheet()** method of **PaymentManager**. The payment sheet UI will persist for 5 minutes after the API is called. If the timer expires, the transaction fails.

```
private PaymentManager.CustomSheetTransactionInfoListener transactionListener =
         new PaymentManager.CustomSheetTransactionInfoListener() {
   @Override
   public void onCardInfoUpdated(CardInfo selectedCardInfo, CustomSheet customSheet) {
      AmountBoxControl amountBoxControl = (AmountBoxControl)
      customSheet.getSheetControl(AMOUNT_CONTROL_ID);
      customSheet.updateControl(amountBoxControl);
      try {
         paymentManager.updateSheet(customSheet);
      } catch (IllegalStateException | NullPointerException e) {
         e.printStackTrace();
      }
   }
```

```
// Receive callback when payment is approved
@Override
public void onSuccess(CustomSheetPaymentInfo response, String paymentCredential,
            Bundle extraPaymentData) {
   try {
      String DPAN = response.getCardInfo().getCardMetaData().getString(SpaySdk.EXTRA_LAST4_DPAN,
                  "Null");
      String FPAN = response.getCardInfo().getCardMetaData().getString(SpaySdk.EXTRA_LAST4_FPAN,
                  "Null");
      Snackbar.make(fragmentView, " DPAN: " + DPAN + " FPAN: " + FPAN,
            Snackbar.LENGTH_LONG).setAction(getString(R.string.ok), new View.OnClickListener() {
               @Override
               public void onClick(View v) {
               }
            }).show();
      } catch (NullPointerException e) {
            e.printStackTrace();
      }
      Toast.makeText(context, "Transaction : onSuccess", Toast.LENGTH_LONG).show();
   }

  // Receive callback if transaction fails
   @Override
   public void onFailure(int errorCode, Bundle errorData) {
      Toast.makeText(context, "Transaction : onFailure : "+ errorCode, Toast.LENGTH_LONG).show();
   }
 };

// Start the transaction request
private void startInAppPayWithCustomSheet() {
  // Show payment sheet
  try {
     Bundle bundle = new Bundle();
     bundle.putString(SamsungPay.PARTNER_SERVICE_TYPE, SamsungPay.ServiceType.INAPP_PAYMENT.toString());
     final PartnerInfo partnerInfo = new PartnerInfo(serviceId, bundle);
     paymentManager = new PaymentManager(context, partnerInfo);
     // Populate the payment sheet
     paymentManager.startInAppPayWithCustomSheet(makeCustomSheetPaymentInfo(),
       transactionListener);
  } catch (NullPointerException e) {
       Toast.makeText(context, SHORTTAG + "Mandatory fields cannot be null.",
          Toast.LENGTH_LONG).show();
       e.printStackTrace();
  } catch (IllegalStateException e) {
       Toast.makeText(context, SHORTTAG + "IllegalStateException", Toast.LENGTH_LONG).show();
       e.printStackTrace();
  } catch (NumberFormatException e) {
       Toast.makeText(context, SHORTTAG +
         "Amount values are not valid",
         Toast.LENGTH_LONG).show();
       e.printStackTrace();
```

```
    } catch (IllegalArgumentException e) {
        Toast.makeText(context, SHORTTAG +
           "Not all mandatory fields set or invalid.",
           Toast.LENGTH_LONG).show();
        e.printStackTrace();
    }
}
```

Shown in Figure 12, when you call the **startInAppPayWithUserInfo()** method, the Samsung Pay payment sheet is displayed as an overlay on the merchant app's screen; from it, the user can select a card for payment and/or change the billing address/shipping address. As discussed earlier (Use Case #2), the result of user interaction with the sheet is delivered through a **CustomSheetTransactionInfoListener** object. When a valid Samsung Pay user authenticates, the **paymentCredential** is returned within the **onSuccess()** event.

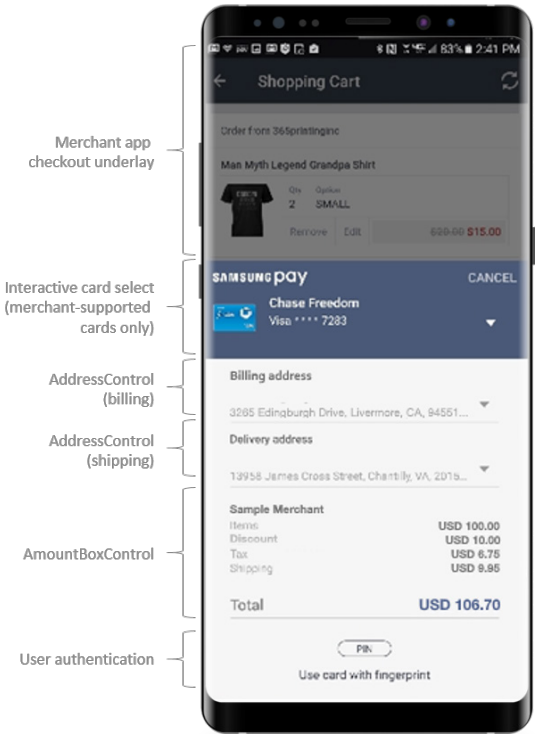

Figure 12. Standard Payment Sheet UI

Handling user-entered changes

SheetUpdatedListener captures the response from the Samsung Pay app whenever the user makes allowed changes on the payment sheet UI. When a change is made, use the **updateSheet()** method to update the payment sheet, even if there is no change in price. You can also use **updateSheet()** when the user changes the shipping address to an invalid or undeliverable address.

The following example demonstrates how to handle a change in the selected card and displays an error message about a shipping address that is ineligible for free shipping.

```
@Override
public void onCardInfoUpdated(CardInfo selectedCardInfo, CustomSheet customSheet) {
    AmountBoxControl amountBoxControl = (AmountBoxControl) customSheet.getSheetControl(AMOUNT_CONTROL_ID);
    if (amountBoxControl == null) {
        Log.d(TAG, "Transaction : Failed / amountBoxControl is null.");
        return;
    }
```

```
    // Update the amount info on payment sheet based on user changes
    amountBoxControl.updateValue(PRODUCT_ITEM_ID, 100);
    amountBoxControl.updateValue(PRODUCT_TAX_ID, 7.45);
    amountBoxControl.updateValue(PRODUCT_SHIPPING_ID, 10);
    amountBoxControl.setAmountTotal(117.45, AmountConstants.FORMAT_TOTAL_PRICE_ONLY);
    customSheet.updateControl(amountBoxControl);
    // Call updateSheet() with a CUSTOM_MESSAGE error code to display a text message.
    paymentManager.updateSheet(customSheet, PaymentManager.CUSTOM_MESSAGE,
         "Delivery address is not eligible for free shipping.");
}
```

Configuring payment sheet controls

The **CustomSheet** object sent in a **startInAppPayWithCustomSheet()** method (Use Case #3) supports four sheet controls — **AddressControl**, **SpinnerControl**, **PlainTextControl**, and **AmountBoxControl**. By contrast, the **CustomSheet** object sent in a **startInAppPayWithUserInfo()** call (Use Case #2) restricts support to **AddressControl** and **AmountBoxControl** only.

AddressControl. This control is used to display the billing or shipping address from Samsung Pay's **My info** user profile, or else the user's addresses provided by the merchant app during the transaction request.

When creating the control, a **controlId** and **SheetItemType** are needed to distinguish the billing address from the shipping address. Otherwise, the merchant app sets the following properties:

- **Address title** – displays a merchant-defined title on the payment sheet. If empty, the default title "Billing address" is displayed.
- **Address** – provides the method to retrieve address details.
- **SheetUpdatedListener** – captures the response from the Samsung Pay app; the merchant app must configure an **AmountBoxControl** to display payment information on the custom payment sheet. When the **onResult()** callback is received, **updateSheet()** must also be called to update the current payment sheet, unless for Use Case #2, sheet updates are turned off with **RequestType.NO_SHEET_UPDATE_CALLBACK**.
- **ErrorCode** – carries error codes directly related to the address.

Here's sample code demonstrating construction of the **AddressControl** for both the billing and shipping addresses. Again, construction of the payment sheet can be handled directly by the merchant app if it adds **samsungpay.jar** as an external library.

```
// For billing address
private AddressControl makeBillingAddressControl() {
    AddressControl billingAddressControl = new AddressControl(BILLING_ADDRESS_ID,
         SheetItemType.BILLING_ADDRESS);
    billingAddressControl.setAddressTitle("Billing Address");
    //This callback is received when control is updated in Samsung Pay
    billingAddressControl.setSheetUpdatedListener(new SheetUpdatedListener() {
        @Override
        public void onResult(String updatedControlId, CustomSheet customSheet) {
            Log.d(TAG,"onResult billingAddressControl updatedControlId: " + updatedControlId);

            // Validate billing address and set errorCode, as needed
            AddressControl addressControl =
                  AddressControl.customSheet.getSheetControl(updatedControlId);
            CustomSheetPaymentInfo.Address billAddress = addressControl.getAddress();
            int errorCode = validateBillingAddress(billAddress);
            Log.d(TAG, "onResult updateSheetBilling errorCode: " + errorCode);
```

```
            addressControl.setErrorCode(errorCode);
            customSheet.updateControl(addressControl);

            // update transaction values if changes are made to address-- mandatory
            /* AmountBoxControl amountBoxControl = (AmountBoxControl)
             * CustomSheet.getSheetControl(AMOUNT_CONTROL_ID);
             * amountBoxControl.updateValue(PRODUCT_ITEM_ID, 1000);
             * amountBoxControl.updateValue(PRODUCT_TAX_ID, 50);
             * amountBoxControl.updateValue(PRODUCT_SHIPPING_ID, 10);
             * amountBoxControl.updateValue(PRODUCT_FUEL_ID, 0, "Pending");
             * amountBoxControl.setAmountTotal(1060, AmountConstants.FORMAT_TOTAL_PRICE_ONLY);
             * customSheet.updateControl(amountBoxControl);
             */

            // Call updateSheet for the full AmountBoxControl; mandatory if updated above
            try {
                 paymentManager.updateSheet(customSheet);
            } catch (IllegalStateException | NullPointerException e) {
                 e.printStackTrace();
            }
        }
    });
    return billingAddressControl;
}

// For Shipping address
private AddressControl makeShippingAddressControl() {
    AddressControl shippingAddressControl = new AddressControl(SHIPPING_ADDRESS_ID,
         SheetItemType.SHIPPING_ADDRESS);
    shippingAddressControl.setAddressTitle("Shipping Address");
    CustomSheetPaymentInfo.Address shippingAddress = new CustomSheetPaymentInfo.Address.Builder()
          .setAddressee("Carol Customer")
          .setAddressLine1("675 East Middlefield Road")
          .setAddressLine2("")
          .setCity("Mountain View")
          .setState("CA")
          .setCountryCode("USA")
          .setPostalCode("94043")
          .setPhoneNumber("555-123-1234")
          .setEmail("carol@provider.com")
          .build();
    shippingAddressControl.setAddress(shippingAddress);
    /*
     * Set address display options for payment sheet; otherwise, default addressControl is displayed
     * Possible values are any combination of these constants:
     *    {DISPLAY_OPTION_ADDRESSEE}
     *    {DISPLAY_OPTION_ADDRESS}
     *    {DISPLAY_OPTION_PHONE_NUMBER}
     *    {DISPLAY_OPTION_EMAIL}
     */
```

```
    int displayOption_val = AddressConstants.DISPLAY_OPTION_ADDRESSEE; // Addressee is mandatory
    displayOption_val += AddressConstants.DISPLAY_OPTION_ADDRESS;
    displayOption_val += AddressConstants.DISPLAY_OPTION_PHONE_NUMBER;
    displayOption_val += AddressConstants.DISPLAY_OPTION_EMAIL;
    shippingAddressControl.setDisplayOption(displayOption_val);
    return shippingAddressControl;
}
```

SpinnerControl. The **SpinnerControl** is used to list various options such as shipping method or installment/subscription plan. It requires a **controlId**, **title**, and **SheetItemType** to distinguish between the spinners displayed (i.e., shipping method or payment plan). The **SHIPPING_METHOD_SPINNER** type can only be used when the shipping address comes from the Samsung Pay app; i.e., when the **CustomSheetPaymentInfo.AddressInPaymentSheet** option is set to **NEED_BILLING_AND_SHIPPING** or **NEED_SHIPPING_SPAY**.

Here's sample code demonstrating how to structure a **SpinnerControl**.

```
// Construct shipping method
SpinnerControlSpinnerControl spinnerControl = new SpinnerControl(SHIPPING_METHOD_SPINNER_ID,
    "Shipping Method ", SheetItemType.SHIPPING_METHOD_SPINNER);
spinnerControl.addItem("shipping_method1", getString(R.string.standard_shipping_free));
spinnerControl.addItem("shipping_method2", getString(R.string.twoday_shipping) );
spinnerControl.addItem("shipping_method3", getString(R.string.oneday_shipping));
// Set default option
spinnerControl.setSelectedItemId("shipping_method1");
// Then listen for SheetControl events
spinnerControl.setSheetUpdatedListener(new SheetUpdatedListener() {
   @Override
   public void onResult(String updatedControlId, CustomSheet customSheet) {
      AmountBoxControl amountBoxControl =
         (AmountBoxControl) customSheet.getSheetControl(AMOUNT_CONTROL_ID);
      SpinnerControl spinnerControl =
         (SpinnerControl) customSheet.getSheetControl(updatedControlId);
      switch (spinnerControl.getSelectedItemId()) {
         case "shipping_method1": amountBoxControl.updateValue(PRODUCT_SHIPPING_ID, 10);
            break;
         case "shipping_method2": amountBoxControl.updateValue(PRODUCT_SHIPPING_ID, 10 + 0.1);
            break;
         case "shipping_method3": amountBoxControl.updateValue(PRODUCT_SHIPPING_ID, 10 + 0.2);
            break;
         default: amountBoxControl.updateValue(PRODUCT_SHIPPING_ID, 10);
             break;
      }
      amountBoxControl.setAmountTotal(1000 + amountBoxControl.getValue(PRODUCT_SHIPPING_ID),
            AmountConstants.FORMAT_TOTAL_PRICE_ONLY);
      customSheet.updateControl(amountBoxControl);
       // Update the payment sheet. Mandatory.
      try {
         paymentManager.updateSheet(customSheet);
      } catch (IllegalStateException | NullPointerException e) {
         e.printStackTrace();
      }
   }
});
```

PlainTextControl. This control is used for displaying a title with two lines of text or a single line of text without a title. A **controlId** is required. Otherwise, the merchant app sets both the title, as appropriate, and a line or two of special informative messaging to the user.

```
private PlainTextControl makePlainTextControl() {
   PlainTextControl plainTextControl = new PlainTextControl("ExamplePlainTextControlId");
   plainTextControl.setText("Just for you:", "Save 10% on orders of $100 or more!");
   return plainTextControl;
}
```

AmountBoxControl. This control is used for displaying purchase amount information on the payment sheet. It requires a **controlId** and a **currencyCode**, and consists of **Item**(s) and **AmountTotal**, defined as follows:

- **Item** – consists of **id**, **title**, **price**, and **extraPrice**. If there is an **extraPrice** in **AmountBoxControl**, its text is displayed on the payment sheet even though there is an actual (numerical) price value. If there is no **extraPrice**, then **currencyCode** with the price value is displayed.
- **AmountTotal** – consists of **price** and **displayOption**. The **displayOption** allows predefined strings only. Your merchant app can set the text to "Estimated amount", "Amount pending", "Pending", "Free", and so forth. The UI format for the string is different for each option.

  Note: The **setAmountTotal** API will accept strings that are not predefined as an argument, although execution will generate an invalid parameter condition and return an error code. It's also important to point out that, depending on the merchant app's preference/policy regarding the transaction details to display in the payment sheet for user authorization, only the transaction's **AmountTotal** may be needed.

Here's how to build the **AmountBoxControl**:

```
private AmountBoxControl makeAmountControl() {
   AmountBoxControl amountBoxControl = new AmountBoxControl(AMOUNT_CONTROL_ID, "USD");
   amountBoxControl.addItem(PRODUCT_ITEM_ID, "Items", 1000, "");
   amountBoxControl.addItem(PRODUCT_TAX_ID, "Tax", 50, "");
   amountBoxControl.addItem(PRODUCT_SHIPPING_ID, "Shipping", 10, "");
   amountBoxControl.setAmountTotal(1060, AmountConstants.FORMAT_TOTAL_PRICE_ONLY);
   amountBoxControl.addItem(3, PRODUCT_FUEL_ID, "FUEL", 0, "Pending");
   return amountBoxControl;
}
```

Important: Call the **updateValue(item_id)** method of **AmountBoxControl** to update each amount item if the item/value changes, then call **CustomSheet.updateControl()** to make the changes take effect in Samsung Pay. Eventually, **PaymentManager.updateSheet(CustomSheet)** must be called to let Samsung Pay know that no further action is pending in the merchant app.

```
   // Update items/values
   amount.updateValue("shippingId", 10);
   amount.updateValue("fuelId", 0);
   amount.setAmountTotal(1000, AmountConstants.FORMAT_TOTAL_PRICE_ONLY);

   sheet.updateControl(amount);
   // Call updateSheet; mandatory whenever values change
   try {
        paymentManager.updateSheet(sheet);
   } catch (IllegalStateException | NullPointerException e) {
        e.printStackTrace();
   }
```

For all **SheetControls**, consult your Javadoc API reference in the **Documents** folder of your SDK package for additional/supplemental guidance.

Sample payment credential

The **paymentCredential** structure will vary depending on the PG's policies/practices currently in place. The following example for a Visa card, representing what is typically sent to a PG in network token (direct) mode — e.g., First Data — is included here strictly to illustrate the general form.


```
{
"billing_address":{"city":"BillingCity","country":"USA","state_province":"CA","street":"BillingAddr1",
"zip_postal_code":"123456"},
"card_last4digits":"1122",
"3DS":{"data":"eyJhbGciOiJSU0ExXzUiLCJraWQiOiJCak91a1h2aFV4WU5wOFIwVGs2Y25OaCtZWWFqZXhIeHRVZ0VFdHlhYy9
NPSIsInR5cCI6IkpPU0UiLCJjaGFubmVsU2VjdXJpdHlDb250ZXh0IjoiUlNBX1BLSSIsImVuYyI6IkExMjhHQ00ifQ.Fg2OOUvHdG
KkIVyBa2S5KtUrPWUeujKZEyxz7n6kALhQahszv3P5JaBaOJ-
RoKcznFjDg3qierzjktU7zXST9gwv4Oclahpfdw64w0X6TtAxeYJiIVkJUG-
edXXTWaJeyeIkgC68wEhF1CltSqG4zLWi6upVCAywdPpBN0Hl0C5WcF5Az4WABYtV_Fda5aHGuyPnE70kEQRTWdlacW9MzEJx2Xth7
Msd9OHoulR8LUQ-7gha17jHoOBwgMoQ9q0hAoCNm0LjWiuhKoRyyu-
Njulnbkk8FZus_AIuMgdv2YN9ygFqIlMculb0VWuF0YeKX6IsgAxi0ZQhLiUsJkCZ_w.AuZZxoG46lnrtk3Q.QE2llwS30VzH-
ZduuE8b045CnfRm2p-RjZGBnZcHELS3v26N64cFg1AV5mtP5f-
fSwbJ3ntP5x4V1NK8FmdY0uSPxzeMfvl5badGAC7w9FrXt6X5xV1Fqu6-q-
ZkbxcB9bYgownt983BcKOE1bd5djxFBOdLrc4j68ikDjc5M3LEBDx6hV0aQzKmilCH-
JeVL3AwQyKBny4Vj7m3Fizw7u1PRLI2ZfWUkXDfS4Vwv3bPm4QUDEMVnHXJ.qTYmdmn4ne93juljNmWkJg","type":"S","versio
n":"100"},
"merchant_ref":"MerchantId",
"method":"3DS",
"recurring_payment":false
}
```


Decrypt using the merchant's private key.


```
-----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEA4LZYjQR+dqd/XLEOXct9jwTJXHD2PTJke9djtMIjKi0h2Oc2GHoW4uJHHY/1jvFt2+zCnjTOXuVLp+76/
DWA3bCwFRj+fPP6x5KKYlPb+dJDYo1TTumltNqCWymJB3u7jBC+xR4vKfRzqjxkE7xhN/SBb82uE8c3sMzVKYnUJi<…>
-----END RSA PRIVATE KEY-----
```


The decrypted result will look similar to the following example. See Samsung Pay token data for other items that can be included.


```
{
   "amount":"1000",
   "currency_code":"USD",
   "utc":"1490266732173",
   "eci_indicator":"5",
   "tokenPAN":"1234567890123456",
   "tokenPanExpiration":"0420",
   "cryptogram":"AK+zkbPMCORcABCD3AGRAoACFA=="
}
```


In gateway token (indirect) mode — e.g., Stripe, Braintree — **paymentCredential** is the PG's token reference ID and its status. Here is an example of the JSON output.


```
{
   "reference":"tok_18rje5E6SzUi23f2mEFAkeP7",
   "status":"AUTHORIZED"
}
```

The merchant app should be able to pass this token object directly to **Charge** or another appropriate payment processing API your PG SDK provides in support of gateway tokens.

Testing and validating your SDK-SDK integration

Testing your integrated SDK for merchant aggregation is crucial in validating Samsung Pay transaction performance and ensuring a positive payment experience for your merchant partners and their customers. Over the course of the validation process, it's important to remember that the goal of testing is not merely to find errors and bugs but to fully understand the quality of your integrated solution.

Testing prerequisites

Given that all PG partners are somewhat different, Samsung Pay can only prescribe a general strategy for merchant aggregation testing, execution, and management in the form of "Best Practices." The general guidelines offered here are designed to furnish a baseline for testing and acceptance from which all partners can benefit. With that goal in mind, the following prerequisites/preconditions should be in place:

- At least one sample merchant app integrated with the PG SDK and Samsung Pay SDK added as third-party libraries.
- At least one Samsung Pay eligible test device plus one ineligible device, both supporting a sample merchant app to test Samsung Pay app status retrieval accuracy.
- One or more Samsung Pay test accounts
- One or more payment cards for each card brand supported by the PG, enrolled under the Samsung Pay test accounts.
- PG sandbox connectivity to validate gateway and network token processing and reporting.

Note: In your sandbox environment, have Samsung Pay return valid testing nonces that point to dummy account data, then make sure that the card selected in Samsung Pay is not charged.

General test conditions and objectives

The objective of any test is to verify the functionality being examined and validate the result produced. All tests should execute and verify test scripts, identifying, fixing, and then retesting all high and medium severity defects in accordance with specific test criteria. Here are some general test conditions and the expected result:

Condition/Criteria	Expected Result
Device ineligible	No Samsung Pay button displayed
Device eligible, Samsung Pay app active ("ready"), eligible payment card present	Samsung Pay button displayed
Device eligible, Samsung inactive ("not ready")	Samsung Pay button displayed but not activate for tapping; message should read: "Please install/activate the Samsung Pay app to continue."

Recommended test cases

At a minimum, the following best practices are recommended for testing your app's integration with Samsung Pay:

- Test with the Samsung Pay app present (installed) on the device, then not installed on device
- Test when the Samsung Pay app is registered and active with enrolled cards present, with no enrolled cards present, and then test when the Samsung Pay app is present but inactive (no registered user)
- Test with supported cards present (enrolled), then with no supported cards present
- Test transactions with all PG-supported card brands

- Validate correct display of user profile information and payment information, if configured
- Test correct display and performance of payment sheet controls (**AmountBoxControl**, **AddressControl**), when applied.
- Test user card selection in payment sheet
- Test accuracy of user-entered edits and changes
- Validate all results received by and from the aggregator PG.

Collecting and sending device dump state (SYSDUMP) logs

To collect a log, complete the following steps:

1. Reproduce the issue.
2. Launch the **Phone** dialer app and dial ***#9900#**.
3. From the menu, select **RUN DUMPSTATE**.
   (Generating the log will take a few minutes. Be patient.)
4. From the **Dump Result** message, note the file name and saved location (typically, **/data/log/<filename>.log**).
5. Copy the log to your PC via USB connection, then zip and send to Samsung Pay.

If you are unable to see or copy the log file, select **COPY TO SDCARD** during #5 above.

Deployment

The decision to release for merchant adoption will be made jointly by the PG with Samsung Pay in accordance with governing agreements.

Branding

Branding specifications and display guidelines are available at https://pay.samsung.com/developers/resource/brand. You can also click **Branding guidelines** under **RESOURCES** after signing in. Here, you can download the full *Branding Guide* for Samsung Pay, as well as the collection of approved assets.

About Samsung Electronics Co., Ltd.

Samsung Electronics Co., Ltd. is a global leader in technology, opening new possibilities for people everywhere. Through relentless innovation and discovery, we are transforming the worlds of televisions, smartphones, personal computers, printers, cameras, home appliances, LTE systems, medical devices, semiconductors, and LED solutions. We employ more than 250,000 people across 79 countries with annual sales exceeding KRW 240 trillion. To discover more, please visit www.samsung.com.

For more information about Samsung Pay, visit http://www.samsung.com/us/samsung-pay/.

Samsung Electronics Co., Ltd.
416, Maetan 3-dong, Yeongtong-gu
Suwon-si, Gyeonggi-do 443-772, Korea

DISCLAIMER

With respect to this "PG Merchant Aggregator Onboarding Guide for In-App Payments and Account Preprovisioning" and any other documents available from this site or server, neither Samsung nor any of its affiliates or employees makes any warranty, express or implied, including the warranties of merchantability and fitness for a particular purpose, or assumes any legal liability or responsibility for the accuracy, correctness, completeness or usefulness of any information, product, technology or process disclosed.