Implementation Guide for CardConnect Consumer SDK

**Tokenizing Card Data with custom UI through CardConnect’s Card Secure gateway:**

In order to be able to tokenize card information the SDK needs to have access to the Card Secure gateway to consume the tokenization service, therefore an URL needs to be provided by CardConnect to the application implementer:

In order to set this URL implementer should call the CCConsumer class which provides a Singleton to facilitate communication with the Consumer API:

|  |
| --- |
| CCConsumer.*getInstance*().getApi().setEndPoint(**"https://url/to/tokenize/data"**); |

1. Secondly in order to call the tokenization API service an implementation of CCConsumerTokenCallback needs to be provided to listen for events from the Consumer API:

|  |
| --- |
| **public class** MainActivity **extends** AppCompatActivity **implements** CCConsumerTokenCallback {  @Override  **protected void** onCreate(Bundle savedInstanceState) {  **super**.onCreate(savedInstanceState);  setContentView(R.layout.***activity\_main***);  }   @Override  **public void** onCCConsumerTokenResponseError(@NonNull CCConsumerError ccConsumerError) {  *//An error occurred trying to tokenize the data  //Notify user of error* }   @Override  **public void** onCCConsumerTokenResponse(@NonNull CCConsumerAccount ccConsumerAccount) {  *//Populate third party server request and send api call to proceed with the authorization.* } } |

1. Populate information to CCConsumerCardInfo object in order to proceed with sending the object to CardSecure gateway and tokenize the data.

There are two ways of populating this object:

* 1. Populate card data information with Consumer SDK custom UI components:

CCConsumerCreditCardNumberEditText

CCConsumerCvvEditText

CCConsumerExpirationDateEditText

If used, these components provide format options that are configurable. Check code documentation for more information.

Card information through these components are not going to be accessible from the implementer side, therefore the implementer needs to pass the CCConsumerCardInfo object to every component in order to populate it internally. Check the Demo application for an example of how to populate this object.

* 1. Populate CCConsumerCardInfo object manually:

In this case, real card information should be set in the object. Check Documentation for more information.

1. Process API Token generation:

After CCConsumerCardInfo object is populated, generateTokenWithCard API call should be made:

|  |
| --- |
| CCConsumer.*getInstance*().getApi().generateAccountForCard(**mCCConsumerCardInfo**  **this**); |

**Tokenizing Card Data with card reader functionality through CardConnect’s Card Secure gateway:**

Tokenization generation occurs internally within the consumer SDK together with the card reader implementation. In order to get the token generated by the card reader an implementation of SwiperControllerListener interface must be provided by the implementer.

The following snippet shows how to instantiate an implementation of the SwiperControllerListener:

|  |
| --- |
| SwiperControllerListener swiperControllerListener = **new** SwiperControllerListener() {  @Override  **public void** onTokenGenerated(CCConsumerAccount ccConsumerAccount, CCConsumerError ccConsumerError) {  *//Dismiss loading indicator in case was shown from onStartTokenGeneration()  //Check ccConsumerError object != null first in case there was an error.  //Populate third party server request and send api call to proceed with the payment.* }   @Override  **public void** onError(SwipeError s) {  *//SwiperControllerListener callback: A Toast or some sort of other visual indicator can be displayed in the  //event of a reader error or other transaction-related error.* }   @Override  **public void** onSwiperReadyForCard() {  *//SwiperControllerListener callback: It is recommended that a visual indicator be shown here  //notifying that the reader is ready for a card swipe.* }   @Override  **public void** onSwiperConnected() {  *//SwiperControllerListener callback: It is recommended that a visual indicator be shown here  // notifying that the reader was connected* }   @Override  **public void** onSwiperDisconnected() {  *//SwiperControllerListener callback: It is recommended that a visual indicator be shown here  // notifying that the reader was disconnected.* }   @Override  **public void** onStartTokenGeneration() {  *//Should use this callback to show some sort of loading indicator.* }  @Override **public void** onBatteryState(BatteryState batteryState) {  *//Should use this callback to notify the user swiper is running out of battery* }  };  } |

A SwiperController object is used to initiate communication between the card reader and payment application. Inside of the onCreate() method of the payment Activity, instantiate a SwiperController object, supplying the SwiperControllerListener implementation:

|  |
| --- |
| SwiperController **mSwiper**; @Override **protected void** onCreate(Bundle savedInstanceState) {  **super**.onCreate(savedInstanceState);  setContentView(R.layout.***activity\_main***);   *//Initialize the swiper using the CCSwiperControllerFactory.* **mSwiper** = (**new** CCSwiperControllerFactory()).create(**this**, SwiperType.***BBPosDevice***, **mSwiperControllerListener**);  *//Optional enable debugging to print debug info generated inside the swiper class* **mSwiper**.setDebugEnabled(**true**); } |

When the Activity corresponding to the payment screen is destroyed, the card reader resources must be cleaned up. In the Activity’s onDestroy() lifecycle callback method, cleanup the resources.

|  |
| --- |
| @Override **protected void** onDestroy() {  **super**.onDestroy();  *//release() must be called in the Activity life cycle onDestroy() method* **mSwiper**.release(); } |

**Using Payments UI flow:**

The Consumer SDK also provides a built-in UI flow for account handling which allows the following operations:

* List all the consumer accounts.
* Delete specified consumer account.
* Update specified consumer account.
* Create specific account with token generation either to manual entry or MSR.

Note: This UI flow is configurable so the implementer can specify format options and UI elements' theme configuration as well.

In order to integrate this UI custom flow implementer should follow these steps:

1. Declare Built-in Activities in consumer application. This will allow the consumer application to specify a desired theme configuration.

|  |
| --- |
| *<!--Declare Consumer Sdk Activities in order to be able to change themes-->* <**activity android:name="com.cardconnect.consumersdk.views.payment.accounts.PaymentAccountsActivity"  android:theme="@style/ConsumerAppImplementer.Theme"** /> <**activity android:name="com.cardconnect.consumersdk.views.payment.createaccount.CreateAccountActivity"  android:theme="@style/ConsumerAppImplementer.Theme"** /> <**activity android:name="com.cardconnect.consumersdk.views.payment.editaccount.EditAccountActivity"  android:theme="@style/ConsumerAppImplementer.Theme"** /> |

|  |
| --- |
| <**style name="ConsumerAppImplementer.Theme" parent="AppConsumerTheme.NoActionBar"**>  *<!--//****TODO Override Attributes****-->*  </**style**> |

1. Declare a specific theme that inherits from AppConsumerTheme.NoActionBar

Note: For more specific information regarding Theme configuration attributes check Theme configuration resources.

1. Implement CCConsumerApiBridge interface class in order to allow the Payments UI flow to populate and make operations with the accounts:

|  |
| --- |
| **public class** ApiBridgeImpl **implements** CCConsumerApiBridge, Parcelable {  *//Parcelable implementation required for passing this object to Consumer SDK* **public static final** Creator<ApiBridgeImpl> ***CREATOR*** = **new** Creator<ApiBridgeImpl>() {  @Override  **public** ApiBridgeImpl createFromParcel(Parcel in) {  **return new** ApiBridgeImpl(in);  }   @Override  **public** ApiBridgeImpl[] newArray(**int** size) {  **return new** ApiBridgeImpl[size];  }  };   **public** ApiBridgeImpl() {  }   **protected** ApiBridgeImpl(Parcel in) {  *//unused* }   @Override  **public void** getAccounts(@NonNull **final** CCConsumerApiBridgeCallbacks apiBridgeCallbacks) {   **final** CCConsumerApiBridgeGetAccountsResponse response = **new** CCConsumerApiBridgeGetAccountsResponse();  *//****TODO Implement get Accounts from Third party server here***  *//****TODO provide result through apiBridgeCallbacks object***  }   @Override  **public void** saveAccountToCustomer(@NonNull **final** CCConsumerAccount account,  @NonNull **final** CCConsumerApiBridgeCallbacks apiBridgeCallbacks) {   **final** CCConsumerApiBridgeSaveAccountResponse response = **new** CCConsumerApiBridgeSaveAccountResponse();  *//****TODO Implement add Account to Profile from Third party server here***  *//****TODO provide result through apiBridgeCallbacks object***  }   @Override  **public void** deleteCustomerAccount(@NonNull CCConsumerAccount accountToDelete,  @NonNull **final** CCConsumerApiBridgeCallbacks apiBridgeCallbacks) {  **final** CCConsumerApiBridgeDeleteAccountResponse response = **new** CCConsumerApiBridgeDeleteAccountResponse();  *//****TODO Implement remove Account to Profile from Third party server here*** *//****TODO provide result through apiBridgeCallbacks object***  }   @Override  **public void** updateAccount(@NonNull CCConsumerAccount account,  @NonNull **final** CCConsumerApiBridgeCallbacks apiBridgeCallbacks) {  *//****TODO Implement update Account to Profile from Third party server here***  *//****TODO provide result through apiBridgeCallbacks object***  }   @Override  **public int** describeContents() {  **return** 0;  }   @Override  **public void** writeToParcel(Parcel dest, **int** flags) {  *//unused* } } |

1. Start PaymentsActivity and pass CCConsumerApiBridge implementation object:

|  |
| --- |
| ApiBridgeImpl apiBridgeImpl = **new** ApiBridgeImpl(); Intent intent = **new** Intent(**this**, PaymentAccountsActivity.**class**); intent.putExtra(PaymentAccountsActivity.***API\_BRIDGE\_IMPL\_KEY***, apiBridgeImpl); startActivityForResult(intent, PaymentAccountsActivity.***PAYMENT\_ACTIVITY\_REQUEST\_CODE***); |

In addition to theme attributes, mask options can be specified through CCConsumerCardFormatter together with API Bridge implementation class.

|  |
| --- |
| CCConsumerCardFormatter formatter = **new** CCConsumerCardFormatter(); formatter.setCCConsumerMaskFormat(CCConsumerMaskFormat.***CARD\_MASK\_LAST\_FOUR***); formatter.setMaskCharacter(**'\*'**); formatter.setCCConsumerExpirationDateSeparator(CCConsumerExpirationDateSeparator.***DASH***); formatter.setCCConsumerCardMaskSpacing(CCConsumerCardMaskSpacing.***CARD\_MASK\_SPACING\_EVERY\_CHARACTER***);  intent.putExtra(PaymentAccountsActivity.***CARD\_FORMAT\_OPTIONS\_KEY***,formatter); startActivityForResult(intent, PaymentAccountsActivity.***PAYMENT\_ACTIVITY\_REQUEST\_CODE***); |

1. Receive Account from Payments UI if selected:

|  |
| --- |
| @Override **protected void** onActivityResult(**int** requestCode, **int** resultCode, Intent data) {  **super**.onActivityResult(requestCode, resultCode, data);   *//Get Account selected by integrated UI flow* **if** (requestCode == PaymentAccountsActivity.***PAYMENT\_ACTIVITY\_REQUEST\_CODE*** && resultCode == ***RESULT\_OK***) {  *//Example of displaying account selected* CCConsumerAccount selectedAccount =  data.getParcelableExtra(PaymentAccountsActivity.***PAYMENT\_ACTIVITY\_ACCOUNT\_SELECTED***);  Log.*d*(***TAG***, **"Selected Account: "** + selectedAccount.getAccountType().toString() + **", "** +  selectedAccount.getLast4());  } } |