CommerceDriver™

Quick-Start Guide for *Android*™

EVO CommerceDriver™

Adding EMV transaction processing to your POS system is easy with the pre-certified EVO CommerceDriver™ SDK. The pre-certified CommerceDriver™ SDK installs alongside your software application to add EMV transaction processing to your POS system. CommerceDriver™ facilitates all transactional communication with the EVO Payments International global processing platforms and approved hardware devices to isolate payment data and keep it separate from the software application.

CommerceDriver™ is designed to support multiple terminal manufacturers while retaining a common API. At startup, CommerceDriver™ detects the supported terminal manufacturer(s)/models for processing Authorize, Authorize & Capture, and Return transactions.

How It Works

1. Create transaction data objects in your POS.
2. Pass the transaction data to CommerceDriver™.
3. CommerceDriver™ initiates terminal commands and gathers tender/EMV data to send to the EVO Snap* Platform.
4. The EVO Snap* Platform returns a response to CommerceDriver™ with receipt details.

Version Details

* CommerceDriver™ - v2.30.X
* Supports EVO Snap* v2.1.30 Platform calls
* Supported Terminals – Ingenico ICMP via Bluetooth, Ingenico iPP320/iPP350 (Ethernet), EVO ITM100 (Audio Jack), and EVO ITP200/ITP250

Compatibility

* CommerceDriver™ Framework – Android API Level 19 (4.4 KitKat)

Google Play Services

Google Play Services is required for Commerce Driver to work.

Check for Google Play Services in your application using GoogleApiAvailability.

Integration

Add to Project

To use CommerceDriver™ in Android, you must first add CommerceDriver™ to your project:

```
dependencies {

    implementation 'com.evosnap.android:commerce-driver:2.29.0'

    // OR

    // inspect pom.xml for additional dependencies
    implementation project(path: 'path-to-commerce-driver)
}
```

Initialize Instance

```
CommerceDriver commerceDriver = new CommerceDriver("applicationProfileId", "serviceKey");
```

Authentication

First Login

Prior to using CommerceDriver™ in full, the user must do the following:

* For a first time login with a temporary password, use `CommerceDriver.changePassword(String, String, String)` method with the username, temporary password, and a new password.
* Then, use the `CommerceDriver.loginWithUsernameAndPassword(String, String)` with your new password.

Subsequent Login

On the first and subsequent login without a temporary password, there are a few account related calls one should make to finish setting up the account:

* Check the password expiration with `CommerceDriver.getUserExpiration(String, String)`
* Check if the user has answered security questions with `CommerceDriver.getSecurityQuestions()`
* If security questions haven't been answered, get the list of available questions with `CommerceDriver.getAvailableSecurityQuestions()`
* Set up security questions with `CommerceDriver.updateSecurityQuestions(String, String, List<SecurityAnswer>)`

Login

```
try {
    CommerceDriver commerceDriver = new CommerceDriver("applicationProfileId", "serviceKey");
    LoginResponse response = commerceDriver.loginWithUsernameAndPassword("username", "password");
    // Success
} catch (SnapSessionError e) {
    // Something went wrong
} catch (SnapApiError e) {
    // An API error occurred
    ApiResponse error = e.getErrorResponse();
    // Look at the error to see what happened
} catch (SnapConnectionError e) {
    // A network problem occurred
} catch (SnapSyncAccountError e) {
    // A problem occurred when syncing your account
}
```

Security Questions

After login, it is important to make sure that security questions have been answered.

Security questions are used to recover forgotten or lost passwords. It is still possible to recover a password if security questions are not set, however this process is longer and requires contacting a customer support representative.

Check if Security Questions Have Been Answered

Use `CommerceDriver.getSecurityQuestions()` to get a list of questions that the user has answered.

If the user has not answered a sufficient number of security questions, (e.g. 3), then they should be prompted to answer security questions.

```
try {
    SecurityQuestionsResponse response = CommerceDriver.getSecurityQuestions();
    List<SecurityQuestion> questions = response.getQuestions();
    if (questions.size() >= RECOMMENDED_SECURITY_QUESTIONS_ANSWERED) {
        // user has answered an appropriate number of security questions
    } else {
        // user should be prompted to answer security questions
    }
} catch (SnapConnectionError e) {
    // a connection error occurred, maybe the network is down or similar
} catch (SnapApiError e) {
    // an api error occurred, maybe the user credentials were invalid
}
```

Get Available Questions to Answer

If the user needs to answer questions, the list of questions can be retrieved via `CommerceDriver.getAvailableSecurityQuestions()`:

```
try {
    SecurityQuestionsResponse response = CommerceDriver.getAvailableSecurityQuestions();
    List<SecurityQuestion> questions = response.getQuestions();
    if (questions.isEmpty()) {
        // something went wrong on the platform
    } else {
```

```
        // offer the security questions to the user to answer
    }
} catch (SnapConnectionError e) {
    // a connection error occurred, maybe the network is down or similar
} catch (SnapApiError e) {
    // an api error occurred, maybe the user credentials were invalid
}
```

Answer Security Questions

Once the user has selected security questions and answers, then `CommerceDriver.updateSecurityQuestions(String, String, List<SecurityAnswer>)` can be called to update:

```
try {
    List<SecurityAnswer> answers = // create a list of answered security questions for password retrieval
    UpdateSecurityQuestionsResponse response = CommerceDriver.updateSecurityQuestions("some_user", "some_pas
sword", answers);
    // security questions have been answered if no exception is thrown
} catch (SnapConnectionError e) {
    // a connection error occurred, maybe the network is down or similar
} catch (SnapApiError e) {
    // an api error occurred, maybe the user credentials were invalid
}
```

Password Expiring Soon

The user password expiration should be checked at each login. The password can be changed at any time, but if the password is expiring relatively soon, then follow these steps:

* Offer the user to change the password via `CommerceDriver.changePassword(String, String, String)`.
* Logout with `CommerceDriver.logout( )`.
* Re-login with `CommerceDriver.login(String, String)` using the changed password.

Terminal Setup

Add a Terminal

Use `CommerceDriver.addTerminal(Terminal)` to add a terminal. If the terminal was added successfully, the method will return `true`. Terminals are identified with an `id`, so each time a new terminal is added, it must have a unique id or the `CommerceDriver.addTerminal(Terminal)` will return `false`.

`Terminal` objects are created using separate terminal libraries.

Multiple terminals may be added.

```
boolean added = commerceDriver.addTerminal(aTerminal);
if (added) {
    // success
} else {
    // failure - a terminal with the given id may have already been added
}
```

Initialize The Terminal

Initialization is required prior to using a terminal. After adding a terminal with `CommerceDriver.addTerminal(Terminal)`, a call to `CommerceDriver.initializeTerminal(InitializeTerminalRequest)` should be made to initialize the terminal.

No other requests to the terminal can be made if the initialization call is not made:

```
commerceDriver.initializeTerminal(myInitializeTerminalRequest);
```

Selecting Terminals

Use `CommerceDriver.selectTerminal(String)` to select a terminal after it has been added. If multiple terminals were added with `CommerceDriver.addTerminal(Terminal)` then this method is how one would change terminals.

Using Terminals

Requests to terminals have a listener as part of the request to receive the results of the request. See below for available requests to the terminal:

Check the Battery

If a terminal has a battery, then the `CommerceDriver.checkBatteryStatus(CheckBatteryRequest)` may be called to check the battery level.

Check the Connectivity

To check if a connection can be made to the terminal, use the `CommerceDriver.checkTerminalConnection(CheckConnectionRequest)` method.

Printing

Printing receipts can be called by creating a `PrintReceiptRequest` which returns a `PrintReceiptResponse`. Not all terminals support printing.

Cancelling

Terminal requests may be flagged for cancellation with `CommerceDriver.cancelRequest()`.The request and the current state of the request will dictate if a cancellation is honored.

Shutting Down

To safely close a terminal connection and instance, a call to `CommerceDriver.shutdownTerminal(ShutdownTerminalRequest)` should be made. If the request succeeds, then the terminal should no longer be used, and the terminal should be removed from CommerceDriver™ via `CommerceDriver.removeTerminalById(String)`.

Terminal Service Management (TSM)

The `InitializeTerminal` method of the CommerceDriver™ object now provides information if an update is available for the terminal currently in use. Users must be signed on to their instance of CommerceDriver™ in order for the initialize terminal process to begin and for the terminal to begin checking for updates. This sign on procedure can be found for each operating system in their respective Quick Start Guides. After performing the steps to authenticate and add a terminal, check the response from the `initializeTerminal()` method to determine if updates are available.

Code Snippets

To initialize a terminal in Java, call the `initializeTerminal()` method in CommerceDriver™. This method runs asynchronously and returns an `ConnectionResult`, which contains a Boolean `isUpdateAvailable()` as well as a list of available updates via `getUpdates()`.

Example code to call and handle the completion of the `initializeTerminal()` method can be found below:

```
class TerminalConnectCallable implements Callable<ConnectResponse> {

    private final CommerceDriver commerceDriver;

    public TerminalConnectCallable(CommerceDriver commerceDriver) {
        this.commerceDriver = commerceDriver;
    }

    @Override
    public ConnectResponse call() throws Exception {
        return commerceDriver.connectTerminal();
    }
}

// MainActivity
 @Override
 public void connectSelectedTerminal() {
          TerminalConnectCallable callable = new TerminalConnectCallable(commerceDriver);
        CallableTask<ConnectResponse> task = new CallableTask<ConnectResponse>(callable);
        task.setUiCallback(this);
        task.setResultCallback(new CallableTaskResultCallback<ConnectResponse>() {
            @Override
            public void onReturnException(Exception exception) {
                // Exception handling here
            }

            @Override
            public void onReturnResult(ConnectResponse response) {
                if (response.isUpdateAvailable()) {
                   // Terminal has an update. Can prompt dialog here and move on, or start update
flow.
                }
                // Connected with no updates available.
            }
        });
        task.execute();
}
```

InitializeTerminalResult

```
package com.evosnap.commercedriver.terminal;

import com.evosnap.commercedriver.cws.terminal.AvailableUpdateInfo;

import java.util.List;

public interface ConnectResponse {

    Result getResult();

    String getErrorMessage();

    List<AvailableUpdateInfo> getUpdates();

    boolean isUpdateAvailable();

    enum Result {
        CONNECTED,
        INVALID_TERMINAL_ID,
        SESSION_REQUIRED,
        TERMINAL_ERROR,
    }
}
```

If the `isUpdateAvailable` property of the `TerminalUpdate` object is true, call the `DownloadAndApplyUpdate` method as described below before the terminal update deadline date.

IMPORTANT! If a terminal has not downloaded the available terminal updates by the associated deadline date, the terminal will be deactivated, preventing any future transactions.

For more information on downloading and applying terminal updates, or the TSM feature as a whole, please see the TSM User Guide.

Transaction Processing

Two transaction sets can be processed using CommerceDriver™.

1. Terminal Required Transactions
   * Authorize
   * Authorize and Capture
   * Return Unlinked
2. No Terminal Required Transactions
   * Undo
   * Capture
   * Return by ID

Starting a Transaction

Transactions can be started by calling `commerceDriver.startTerminalTransaction(builder.build())`. The following example demonstrates how to start a transaction.

1. First, create a simple fragment that implements `TransactionEventListener`. This allows for handling of various UI events that may be triggered throughout a transaction.

```
public class TransactionFragment extends Fragment implements View.OnClickListener, TransactionEventLis
tener {

private PosRequest createPosRequest() {
```

2. In v2.27 of CommerceDriver™, authorization types were set in the authorization call `commercedriver.authorize(posRequest)`, but are now simply defined in the `posRequest`, as shown below:

```
            AuthType authType = AuthType.AUTH_AND_CAP;

            PosRequest.Builder builder = new PosRequest.Builder(authType);
```

3. After defining what type of authorization request to send, populate the rest of the `posRequest` using its builder.

```
        builder.setPurchaseAmount(<double>);

        builder.setOrderNumber(<String>);

        builder.setEmployeeId(<String>);

        builder.setLandeId(<String>);

        builder.setReference(<String>);

        builder.setTransactionDateTime(<Date>);

        builder.setTenderType(TenderType.ProcessAsCredit);
```

4. An event listener is added to the `posRequest` object, as well. This will return all UI related events for the transaction, such as a signature or CVV request dialogue. In the sample, the Android fragment responsible for running transactions is passed in:

```
        builder.setTransactionEventListener(this);
```

5. Lastly, return the builder:

```
        return builder.build();

}
```

Now that the `posRequest` has been built, a terminal transaction can be started with CommerceDriver™ elsewhere in the application, such as a Start Transaction button click event.

```
@Override

public void onClick(View v) {

        PosRequest posRequest = createPosRequest();

        commerceDriver.startTerminalTransaction(posRequest)

}
```

Transaction Example – Process as Debit

Most transactions will look primarily the same with the minor differences shown in building the `posRequest`.

In this instance, to process as PIN Debit, the `TenderType` enum is changed to reflect this:

```
    AuthType authType = AuthType.AUTH_AND_CAP;

    PosRequest.Builder builder = new PosRequest.Builder(authType);

    builder.setPurchaseAmount(<double>);
```

```
builder.setOrderNumber(<String>);

builder.setEmployeeId(<String>);

builder.setLandeId(<String>);

builder.setReference(<String>);

builder.setTransactionDateTime(<Date>);

builder.setTenderType(TenderType.ProcessAsPinDebit);
```

Transaction Example – Return Unlinked

Most components of this transaction type remain unchanged, but the authorization piece is changed from `AUTH_AND_CAP` to `RETURN_UNLINKED`:

```
AuthType authType = AuthType.RETURN_UNLINKED;
PosRequest.Builder builder = new PosRequest.Builder(authType);

builder.setPurchaseAmount(<double>);
builder.setOrderNumber(<String>);
builder.setEmployeeId(<String>);
builder.setLandeId(<String>);
builder.setReference(<String>);
builder.setTransactionDateTime(<Date>);
builder.setTenderType(TenderType.ProcessAsCredit);
builder.setTransactionEventListener(this);

return builder.build();
```

Transaction Data

For the initial implementation, there are only a few pieces of transaction data that should be set. Recall that the user must first declare and initialize a `PosRequestBuilder` (referred to as `builder` below) before calling the methods below. Please see the "Starting a Transaction" section above for more information.

Method	Description
`builder.setAmount(double)`	Sets the total tranasaction amount (including tax, cash back, etc.)
`builder.setCustomerPresent(CustomerPresent)`	Sets the customer presence – most cases will be `CustomerPresent.Present`
`builder.setProcessAsCredit()`	Sets the processing type to "Credit"
`builder.setProcessAsDebit()`	Sets the processing type to "Debit"

`builder.setEmployeeId(String)`	Sets the Employee ID using the POS for the transaction
`builder.setOrderNumber(String)`	Sets the Order ID for the transaction
`builder.setLaneId(String)`	Sets the Lane ID for the transaction
`builder.setReference(String)`	Sets the reference for the transaction
`builder.setTransactionDateTime(Date)`	Sets the transaction date – **Note**: in most cases, `date` can be passed as `newDate( )`
`builder.setTransactionEventListener(TransactionEventListener)`	Sets the Event Listener. More information about Events can be found in the **Event** section below.

Events

As shown in the transaction examples, `TransactionEventListener` works as the observer for both blocking and non-blocking requests to the POS from the `TerminalController`.

Below are the methods that are called on the `TransactionEventListener`:

Method	Description	Recommended Action
`void onRequestSignatureConfirmation(ConfirmSignatureRequest request);`	Called if a signature should be collected and confirmed by the POS	Display a signature dialog and collect signature.
`void onRequestFinalConfirmation(FinalConfirmationRequest request);`	Called if a final confirmation must be made by the POS, as opposed to the terminal	Display a final confirmation dialog with the amount.
`void onRequestApplicationSelection(App`	Called if application selection must be made by the POS, as opposed to the terminal	Display an application selection dialog.

`licationSelectionRequest request);`		
`void onRequestConfirmCardRemoved(ConfirmCardRemovedRequest request);`	Called if the POS should ensure a card has been removed from the terminal	None required
`void onWaitingForCard(EnumSet<CardInterface> cardInterfaces);`	Called when the terminal is ready to read a card through a supported interface, (e.g. contactless, contact chip)	None required
`void onTransactionNotification(TransactionNotification notification);`	Called when an anonymous event occurs during a transaction	None required
`void onCardRead(CardReadData cardReadData);`	Called when card data is read, (e.g. maskedPAN, card type)	None required
`void onRequestApDupeOverride(ApDupeOverrideRequest request);`	Called when a transaction is a duplicate and the POS can override to allow the duplicate transaction to finish processing – this is the default behavior	None required
`void onRequestCVV(CVVResultHandler handler);`	Called when a transaction requires a card CVV to be entered by the cardholder	Display a CVV dialog and return the collected CVV string via the `onCVVEntered(String cvv)` event method provided by the handler.
`void onTransactionCompleted(TransactionResult transactionResult, BankCardTransactionPro request, BankCardTransactionResponsePro response);`	Called when a transaction is completed (approved, declined, cancelled, error, etc.)	Display a dialog with the result and receipt options (e.g. email, print, etc.).

Verify

Verify is a transaction operation added to the CommerceDriver™ which can be used to validate a card. The Verify operation w ill trigger a MSR swipe transaction on the connected terminal with empty transaction fields (no amount, merchantID, etc.).

Creating a Verify Request Example

```
try {
    VerifyRequest builder = PosRequestBuilder.newerifyRequest();
    builder.setTransactionEventListener(myTransactionEventListener);
    commerceDriver.startTerminalTransaction(builder.build());

        // Listen for callbacks!
    } catch (SnapValidationError e) {
        // Commerce Driver didn't like something with the transaction!
    } catch (SnapTerminalError e) {
        // The terminal didn't like something with the transaction!
    } catch (SnapSessionError e) {
        // Your session might be expired! Time to log in again!
}
```

Tokenization

Tokenization is the process of using a token to run what would typically be a card only transaction. The EVO Snap* platform generates a unique token associated with a customer's card that can be used instead of the customer's actual card to process a transaction.

How to Run a Tokenized Transaction

In order to run a tokenized transaction, the `PaymentAccountDataToken` property must be populated with a valid payment token through the transacion builder `` `PosRequestBuilder builder = PosRequestBuilder.newPaymentTokenRequest()` ``. If the PaymentAccountDataToken is populated, CommerceDriver™ will automatically run the token and no card will be needed to process the transaction.

The transaction types that can use payment tokens are listed below:

* Authorize
* Authorize and Capture
* Return Unlinked

Authorize And Capture with Tokenization Example

```
try {
    PosRequestBuilder builder = PosRequestBuilder.newPaymentTokenRequest()
    builder.setAmount(10.00);
    builder.setPaymentAccountDataToken(DATA_TOKEN);
    builder.setTransactionDateTime(new Date());
    builder.authorizeAndCapture();
    builder.setTransactionEventListener(myTransactionEventListener);
```

```
    commerceDriver.startTerminalTransaction(builder.build());

        // Listen for callbacks!
    } catch (SnapValidationError e) {
        // Commerce Driver didn't like something with the transaction!
    } catch (SnapTerminalError e) {
        // The terminal didn't like something with the transaction!
    } catch (SnapSessionError e) {
        // Your session might be expired! Time to log in again!
}
```

Unsuccessful Calls

When executing "terminal" methods, or calls to login, security related calls, etc., it is possible that an error can occur.

Common Exception Reasons

Exceptions can typically occur for the following reasons:

* `SnapConnectionError` is thrown when a network call fails
* `SnapSessionError` is thrown when a session is expired or invalid and/or a login is required
* `SnapApiError` thrown when the platform responds with an API Error
    - `SnapApiError.getErrorResponse()` may provide the error response along with an error code

Common API Error Codes

If a `SnapApiError` is thrown, `SnapApiError.getErrorResponse()` may return an `ApiResponse` containing error details. `ApiResponse.getErrorId()` can return a numeric code indicating the reason for the API Error.

Below are common API Errors when performing any of the calls listed above:

Error ID	Definition	Resolution
406	User credentials invalid	Use valid credentials when calling `CommerceDriver.login(String, String)`
5001	Password change required	Change password using `CommerceDriver.changePassword(String, String, String)`
5002	Account locked – too many invalid logins	Contact your Solutions Engineer

5003	Account locked – Administrative Lock	Contact Snap* Customer Support
5004	Account locked – Password is Expired	Change password using `CommerceDriver.changePassword(String, String, String)`

Reference Information

For additional information, please visit the EVO Snap* Support site at http://www.evosnap.com/support/ or contact your EVO Technical Support representative.