EVO® Snap

CommerceDriver™

Quick-Start Guide for *Windows*®

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™ - v2.29.0
* EVOSnap* Web Services - v2.1.29 (Platform calls)
* Supported Terminal – Ingenico ICMP via Serial USB, Ingenico iPP320/iPP350 via Serial USB, BBPOS Wisepad 2/Wisepad 2+ via Bluetooth

Compatibility

* CommerceDriver™ Framework – Windows® 7+
* Visual Studio 2015
* .Net 4.5

Integration

To get started with CommerceDriver™, select your Platform, Network and Hardware. The setup is similar to a direct Web Services integration, but CommerceDriver™ must be hosted locally.

For our example, the following setup uses the EVOSnap* assembly for interacting with *Ingenico* terminals. The EvoSnap.CommerceDriver.Ingenico.dll assembly must be placed in the same folder as CommerceDriver™.

1. Download the CommerceDriver™ SDK based on Terminal Manufacturer.
2. Uncompress the archive into a temporary folder.
3. Copy the following files into the folders associated with your solution:
   * *EvoSnap.CommerceDriver.Common.dll*
   * *EvoSnap.CommerceDriver.Ingenico.dll*
   * *EvoSnap.CommerceDriver.IntegratedTerminals.dll*
   * *Newtonsoft.Json.dll*
   * *RBA_SDK.dll*
   * *RBA_SDK_CS.dll*
4. Add the assembly references to the solution for the following files:
   * *EvoSnap.CommerceDriver.Common.dll*
5. Place the assemblies below into an associated \bin\debug output folder. (The assemblies above depend on the following assemblies.)
   * *RBA_SDK.dll*
   * *RBA_SDK_CS.dll*
6. Use the (*EvoSnap.CommerceDriver.Common.Controllers) CommerceDriverController* class to create an instance and wire up the default event handlers.

```
// Instantiate the controller and define the logging info
Controller = new CommerceDriverController();
Controller.LogInstanceName = "TC001";
Controller.LogInstanceID = 1;
Controller.LogLevel = LogLevel.Trace;

// Wire up general event handlers
Controller.Log += Controller_Log;
Controller.Notification += Controller_Notification;
Controller.GenerateReceipt += Controller_GenerateReceipt;
Controller.ConfirmSignature += Controller_ConfirmSignature;
Controller.ServiceInvoked += Controller_ServiceInvoked;
Controller.Completed += Controller_Completed;
Controller.RetrieveCvv += Controller_RetrieveCvv;

// Create an instance which contains the default password event handlers
Handlers = new DefaultEventHandlers(Controller);

// Wire up the password specific event handlers using the default ones
Controller.IdentityLogin += Handlers.Default_IdentityLogin;
Controller.ChangePassword += Handlers.Default_ChangePassword;
Controller.PasswordReset += Handlers.Default_PasswordReset;
Controller.ForgotPassword += Handlers.Default_ForgotPassword;
```

```
Controller.AssignQuestions += Handlers.Default_AssignQuestions;
Controller.AccountNotification += Handlers.Default_AccountNotification;
```

Authentication

Call *Login( )* or *LoginAsync( )* in the CommerceDriverController using a username and password:

```
GatewaySession session = CommerceDriver.Login("UserName", "Password");
```

Terminal Setup

1. Initialize the CommerceDriverController:

```
try
{
    // Call the Initialize() method to load the manufacturer/terminal information
    Controller.Initialize();
}
catch (Exception ex)
{
    // An exception can be raised if a DLL or one of its dependancies are not found
    LaunchExceptionDialog(ex);
}
```

2. Based on the terminal brand, use BbposTerminalControllerFactory or IngenicoTerminalControllerFactory to create an ITerminalController object:

```
ITerminalController controller = IngenicoTerminalControllerFactory.CreateIcmpController(ComPort.COM1,
ConnectionTypes.Serial);
```

3. Call *AddTerminal( )* in CommerceDriverController with the ITerminalController object created in Step 2:

```
CommerceDriver.AddTerminal(controller);
```

***Note**: The first terminal added will automatically be set as the SelectedTerminal. The SelectedTerminal can be changed using* SelectTerminal *in CommerceDriverController.*

4. Initialize the terminal by calling *InitializeSelectedTerminal* in CommerceDriverController:

```
bool result = await CommerceDriver.InitializeSelectedTerminal();
```

Transaction Processing

Two transaction sets can be processed using CommerceDriver™.

Terminal Required Transactions

* Authorize
* Authorize and Capture
* Return Unlinked

No Terminal Required Transactions

* Undo
* Capture
* Return by ID

1. Compose a request object to authorize and capture a transaction for $10.00.

```
AuthorizeCaptureOperationRequest request = new AuthorizeCaptureOperationRequest();
request.Amount = 10.00m;
request.EmployeeId = "1234";
request.LaneId = "1";
request.OrderNumber = "7724";
request.Reference = "98106";
request.TipAmount = 0;
request.CashbackAmount = 0;
request.OverrideApDupe = false;
```

2. Invoke the Request.

```
await Controller.ProcessAsync(request);
```

Note: At first invocation, the user is asked to login via the IdentityLogin event. All dialogs presented are from DefaultEventHandlers instance.

3. The Completed event contains the processing results. (Requires successful login and completed transaction processing.)

```
private void Controller_Completed(object sender, CompletedEventArgs e)
{
    switch (e.OperationResponse.Type)
    {
        case OperationType.AuthorizeCapture:
            AuthorizeCaptureOperationResponse response = e.OperationResponse as
AuthorizeCaptureOperationResponse;
```

```
            // response.Request -- Contains the original request
            // response.Responses -- All communication with EvoSnap web services
            // response.TransactionResult -- Approved, Declined etc
            // response.TransactionResponse - Transaction response detail
            break;
        }
    }
```

Verify

Verify is a transaction operation added to the CommerceDriver™ which can be used to validate a card. The Verify operation will cause the terminal to prompt for a MSR card swipe.

CommerceDriverController Method Call

```
Task ProcessAsync(VerifyOperationRequest request)
```

VerifyOperationRequest

```
public class VerifyOperationRequest: TransactionRequest
    {
        #region Constructors

        public VerifyOperationRequest()

        public VerifyOperationRequest(TransactionRequest request) : base(request)

        #endregion

        #region Properties

        public ProcessOperation Operation { get; }

        #endregion

        #region Methods

        public void AddTransactionRequest(TransactionRequest request)

        public string ToString()

        #endregion
    }
```

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 in the `TransactionRequest.CardData` field. If the

`PaymentAccountDataToken` is populated, then 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

Where to Find the Payment Token

In the response of any transaction that uses a card, there will be a field called `PaymentAccountDataToken`. The value in this field is specific to the card that was used in that transaction and can be used to populate `TransactionRequest.CardData.PaymentAccountData` in order to run another transaction for that card.

Transactions that can return a `PaymentAccountDataToken` are listed below:

* Authorize
* Authorize and Capture
* Return Unlinked
* Verify

Default Dialogs/Event Handlers

To simplify the implementation process, default dialogs and associated event handlers are included in the SDK. EVO Snap* highly recommends using the default event handlers for the initial connection to the Platform Services to ensure the password change dialogs are in place. A successful password change is required for the user account to process requests.

The sample code above utilizes the default event handlers, but custom dialogs can be created.

For additional information, please refer to the CommerceDriver™ *Technical Reference Guide for Windows®*.

Core Assemblies

* **EvoSnap.CommerceDriver.Common.dll** – An assembly containing common code and data models as well as the main CommerceDriverController class.

- **Newtonsoft.Json.dll** - An assembly containing code required for serializing/de-serializing JSON models for REST request and responses.

Supplementary Files

- **EvoSnap.CommerceDriver.Extras.dll** – Sample assembly containing shared handlers, helpers and forms

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.