The EdgCapture e-Commerce Option (ECCO) API

Developer's Guide

February 2010

Edgil
an AdStar company

Edgil Associates, Inc.
6 Fortune Drive
Suite 201
Billerica, MA 01821
(978) 262-9799

Contents

Overview

EdgCapture, developed by Edgil Associates in alliance with bank card and check processors, is an integrated PC-based electronic payment processing product tailored specifically for merchants accepting mail, phone, and online orders. EdgCapture performs verification, authorization, and settlement for credit card and checking transactions.

The EdgCapture e-Commerce option (ECCO) provides an interface to the EdgCapture system that allows an application to submit transactions over a secure TCP/IP connection to an EdgCapture system.

Introduction to Payment Processing

Merchants who accept credit cards must interact with several parties in order to receive payment for their goods or services. The parties involved in a credit card transaction include the following:

- The ***merchant*** offers goods or services for sale. In order to accept credit card payments from a Web site or other transaction source where the card and signature cannot be verified, the merchant must have an account with a merchant bank that allows “card not present” or “mail order/telephone order” transactions and a payment processing application, such as EdgCapture.
- The ***customer*** or cardholder makes a purchase from the merchant using a credit card. In order to complete the purchase, the customer must provide personal and account information required by the merchant and the processor.
- The ***issuing bank*** extends a credit card account with a specified credit limit and provides authorization services and monthly statements to the cardholder.
- The ***merchant bank*** (acquiring bank) provides the merchant with an account that allows credit card payments.
- The ***payment network*** (payment processor) moves the transactions submitted by the processing application through the financial networks for authorization and settlement. During authorization, the processor checks the cardholder’s account at the issuing bank. During settlement, the processor transfers the funds from the cardholder’s account to the merchant's account at the acquiring bank. Note that a merchant may use multiple payment networks: one processor for authorization and a second for settlement, or different networks for different credit cards.

The first step in payment processing is authorization. The transaction data is passed to the payment network, which verifies the information and contacts the issuing bank to check that the cardholder's account has adequate funds to cover the purchase. If the account has a sufficient credit balance, the network places a hold on the transaction amount and returns an authorization code to the merchant. Although the authorization hold decreases the customer's available credit, no funds are actually transferred to the merchant's account until the transaction is settled. An authorization hold remains in place for several days. Credit transactions do not need to be sent to the payment network for authorization before they are settled.

During authorization, the payment network may also perform address verification (AVS) and Security Code checking, which reduce the risk involved in "card not present" transactions. Address verification compares the address information transmitted with the transaction to the cardholder's billing address. Security Code checking compares the printed 3- or 4-digit number found on the signature block of most credit cards with the account record.

If the cardholder does not have sufficient funds or if there are other problems with the account, the transaction will be rejected or declined. The payment network may also return a "call center" response, which requires an operator to call for voice authorization. Call center responses are generally triggered by a lost or stolen card or by a change in spending patterns; however, networks also issue random call center responses.

After the transaction has been authorized, the next step is to mark the transaction for capture. According to rules established by the card associations, a transaction may not be captured until the goods purchased have been shipped to the customer. If the product is a service or custom goods, the transaction may be marked automatically after authorization. Depending upon the type of purchase, the period between authorization and capture varies from a few seconds to a few days.

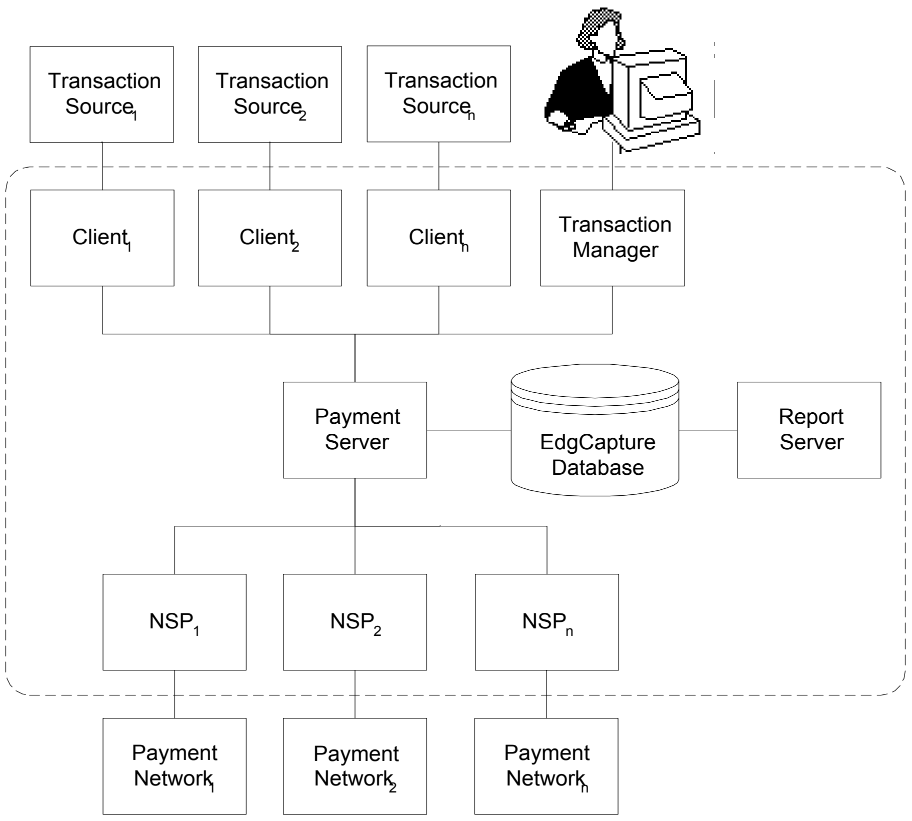
Settlement completes transaction processing. Transactions are settled periodically in groups. When the settlement period arrives, all marked transactions are collected and submitted to the settlement processor, which sends payment instructions to the issuing and acquiring banks. The funds are then transferred electronically between the issuing bank on behalf of the cardholder and the merchant's account. The issuing bank later credits or debits the cardholder's account.

Payment Processing with EdgCapture

EdgCapture is a payment processing system that accepts transactions from multiple sources and provides interfaces to a wide variety of authorization and settlement services.

EdgCapture's design is based on a client-server model. The EdgCapture payment server (credit server) supports a number of EdgCapture client processes that submit transactions originating on a variety of source systems and platforms. All transaction information is stored in EdgCapture's local database. Interactive clients, such as the database client, ECCO, and the Transaction Manager, submit transactions to EdgCapture continuously and receive the processing results in real time. Batch clients accept files of credit card transactions from any source and verify each transaction in the file before submitting them for processing. After processing, batch clients generate a response file, which can be used to update the source system.

The EdgCapture payment server acts as the middleman between the clients and the Network Service Processes (NSPs), which provide standard interfaces to the available payment networks. EdgCapture communicates with the payment networks using a dedicated dial-up or leased telephone line, a frame relay connection, an SSL connection or a secure FTP transfer.



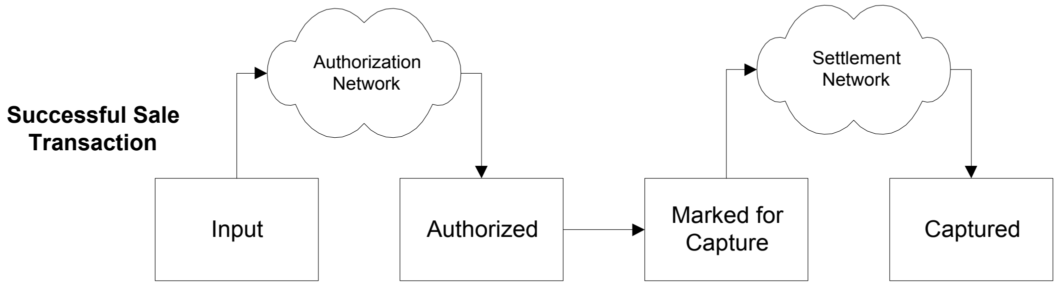
The speed with which EdgCapture processes transactions is limited by the type of connection and the response time of the payment network. Transactions can be fed into the payment server simultaneously through multiple client connections. Multi-threading and multi-tasking are used to keep transactions flowing to the network.

In addition to entering transactions, users of EdgCapture's Transaction Manager can research existing transactions, customers, and accounts, and process call center transactions from a Microsoft Windows PC. EdgCapture includes several standard report formats that can be used to create a variety of specialized reports using the Edgil Report Manager. The EdgCapture system also provides administrative tools for system and user management, as well as extensive error tracking.

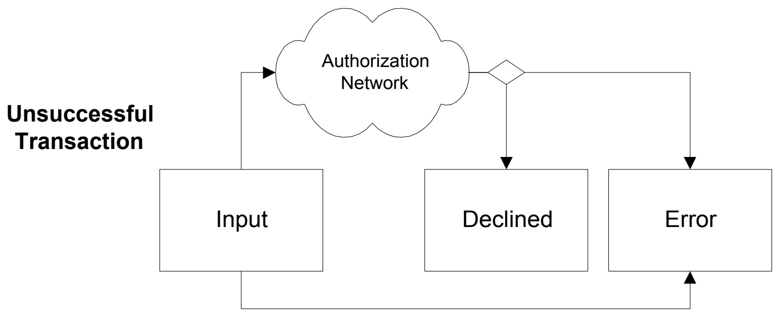
Transaction Flow

As a sale transaction moves through the EdgCapture system, its processing status is indicated by its queue, which is part of the transaction record in the database. When a client submits a transaction to the EdgCapture system, the information is validated and the transaction is placed in the *input* queue. The transaction is then passed to the authorization NSP, which submits it to the appropriate payment network to obtain authorization. If the transaction is authorized, EdgCapture moves the transaction to the *authorized* queue and updates the database with the authorization code. When the client process sends a request to capture the transaction, the transaction moves to the *marked for capture* queue to await settlement. Depending upon its type, a transaction can be marked for capture automatically upon authorization. In this case, EdgCapture moves the transaction directly from *input* to the *marked for capture* queue.

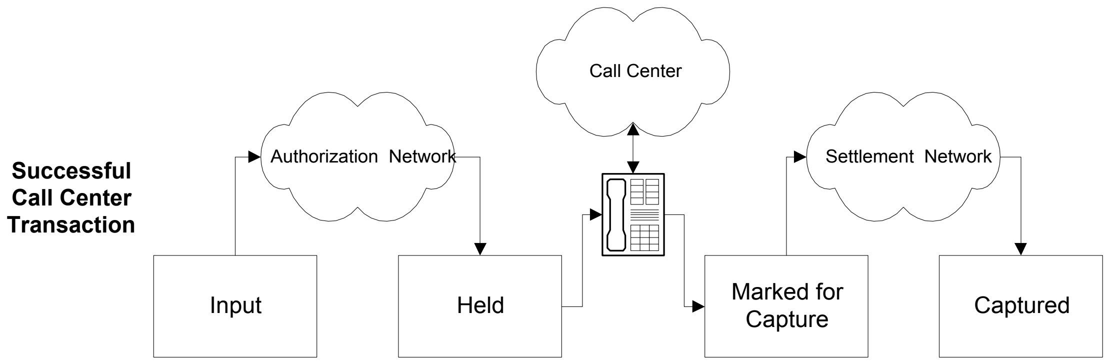
Periodically, EdgCapture creates a batch settlement file containing all transactions marked for capture. The settlement NSP submits the file to the network, which validates the file, accepts the transactions, and initiates the funds transfer. All successful transactions are then moved to the *captured* queue, indicating that they have been sent for settlement.



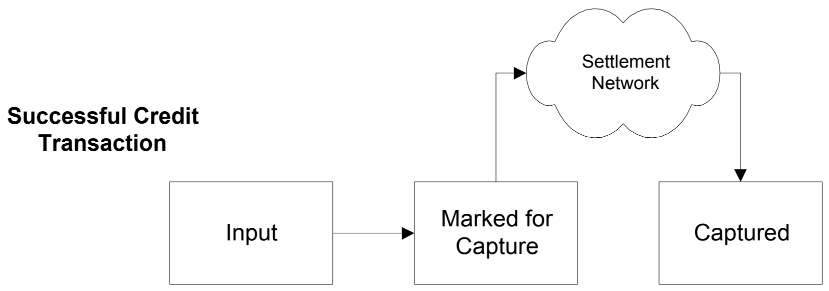
If the transaction is declined, EdgCapture moves the transaction to the *declined* queue. If the network or EdgCapture itself reports an error, the transaction is moved to the *error* queue. Either the client process or a user of EdgCapture's interactive Transaction Manager can resubmit a declined transaction with a different credit card or resubmit a transaction in the error queue after fixing the error.



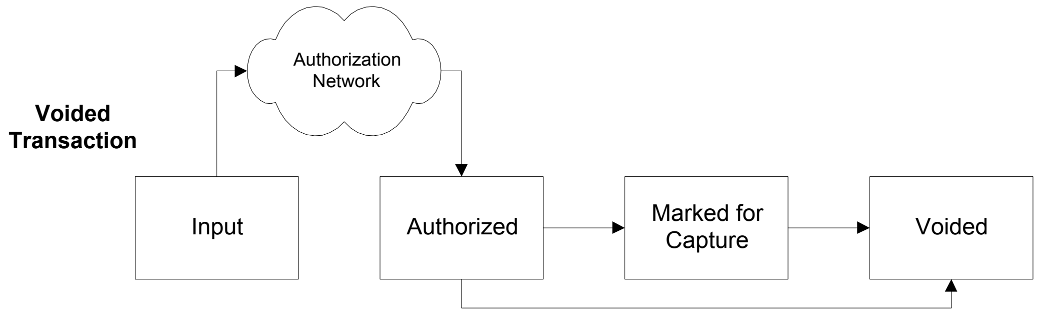
If the network returns a call center response, the transaction is moved to the *held* queue. An EdgCapture user can then call the network for a voice authorization and enter the authorization code or the decline using the Transaction Manager. Voice authorized transactions are moved directly to the *marked for capture* queue to be included in the settlement batch.



Because credit transactions do not need authorization by the payment network, they are moved directly from input to marked for capture. Credits are then included in the settlement batch with the transactions that have been successfully authorized and marked for capture.



EdgCapture also allows the voiding of transactions that have been entered in error. A void transaction must match a previous sale or credit and must take place before settlement of the original transaction. A void prevents the transaction from being settled and from appearing on the customer's statement. Note that a void does not cancel the authorization at the payment network; the hold on the customer's account for the transaction amount remains. EdgCapture moves voided transactions to the *voided* queue.



Transaction Data

EdgCapture identifies transactions as belonging to a particular *merchant* and *order entry process*. An EdgCapture merchant corresponds to a single merchant account at the acquiring bank. Some EdgCapture sites have multiple merchants in order to direct payments to accounts belonging to different departments or business units. An order entry process (OEP) is a subaccount of a merchant, defined by the site for internal accounting, reporting, and security purposes. Most often, an OEP corresponds to a functional area within a business. The OEP may also identify the source of the transaction, for example, the classified order entry system or phone subscription payments. Both merchant and OEP can be used as selection criteria for researching transactions or defining reports.

OEPs also organize the additional information associated with a transaction. Every EdgCapture transaction has 10 additional information fields that can be used for site-specific data. The contents of these fields can vary based on OEP.

Cardholder data is defined as a subset of the transaction data. Cardholder data includes the account number, expiration date, ABA Routing number, and all cardholder address information. A Token can be part of the transaction data. The token specifies a set of cardholder data in Edgcapture. The management of Cardholder data using tokens is referred to as Cardholder Data Management (CDM). See Appendix E for a separate discussion of CDM.

Transaction information includes:

MerchantId	Character or characters assigned to identify a merchant account in the EdgCapture system. REQUIRED
OEPId	Character assigned to identify the order entry process (OEP), or subaccount, of the merchant specified with merchant id. REQUIRED

TransactionId	String that, along with MerchantId and OEPId, uniquely identifies the transaction in the EdgCapture database. REQUIRED
Amount	Amount is a string in the format ‘nnnnn.nn’ with all decimal places expressed. Amount is always positive. REQUIRED
AccountNumber	Credit card or checking account number. REQUIRED
ExpirationDate	Credit card expiration date. REQUIRED for credit card transactions
ABANumber	The American Bankers Association Routing number that identifies the bank issuing a check. REQUIRED for check draft and direct debit transactions
PaymentDesignator	Specifies the type of checking account as “consumer” or “corporate”. REQUIRED for check draft or direct debit transactions.
LastName	Customer’s last name.
FirstName	Customer’s first name.
MiddleInitial	Customer’s middle initial.
AddressLine1	First line of customer’s address.
AddressLine2	Second line of customer’s address.
City	Customer’s city.
StateOrProvince	Customer’s state or province.
ZipCode	Customer’s zip or postal code.
TelephoneNumber	Telephone number.
Notes	Up to 80 characters of notes about the transaction.
UserData1-10	Up to 40 characters of additional data for each of the ten UserData fields.
SalesTax	Amount of sales tax. Required for level II processing for corporate credit cards.
ECI	Electronic commerce indicator to identify the source of transactions.
SecurityCode	3- or 4-digit printed code on the signature plate or front of a credit card.
Token	A 9 character string that can be used in place of cardholder data in a transaction. A token is returned from a monetary transaction request, or a Cardholder Data Management (CDM) request

Introduction to the EdgCapture e-Commerce Option

The EdgCapture e-Commerce Option (ECCO) provides remote interactive payment processing from a merchant’s web or application server over a secure TCP/IP connection. It consists of an additional software module running on EdgCapture that communicates via an XML-based messaging interface with one or more ECCO client applications developed by the merchant using the ECCO API. EdgCapture is capable of handling multiple simultaneous ECCO client connections.

Transactions submitted by the ECCO software are consolidated in the EdgCapture system with transactions from other sources for research and reporting purposes. EdgCapture’s administrative tools let users research transaction information to track account activity, refute chargebacks, and respond to customer inquiries. If necessary, credits and additional charges can be issued from the Transaction Manager by users with access to transactions originating from ECCO. ECCO-based transactions can easily be added to existing reports, including any specially designed reports used to update financial systems.

Features

ECCO’s flexibility allows a site to choose how to respond to situations, such as declined transactions, call center responses, or unavailable payment networks, that occur during payment processing. ECCO also allows an application to retrieve transaction information from the EdgCapture database.

Two-step processing

One characteristic of web-based e-commerce applications, as well as manual order entry on certain front-end systems, is that the customer is willing to wait for only a short period to see whether a purchase has been authorized, but the merchant does not want to deliver the product until authorization has occurred. In addition, the merchant does not want to charge a customer who cancels the transaction after it has been submitted for authorization.

ECCO addresses these issues by using two-step processing. The first step submits the purchase amount for authorization, and the second marks the successfully authorized transaction for capture. If a customer cancels a transaction before the authorization is complete or before the product is delivered, the application can simply not mark the transaction for capture. It remains in EdgCapture’s authorized queue and is never included in the settlement batch.

Manual and automatic processing

Transactions submitted by ECCO can be designated as either manual or automatic, depending upon how the server should operate when the authorization network is temporarily unavailable. If a transaction is submitted as manual, EdgCapture reports an error and does not attempt to request authorization when the connection is reestablished. If the transaction is automatic, EdgCapture continues to retry until it gets a status from the authorization network. The application can later retrieve the transaction information from EdgCapture to determine whether the transaction was authorized.

Security

EdgCapture is PCI compliant and full certified. See PCI Security Standards Council Validated Payment Applications. EdgCapture protects sensitive transaction data at each point during processing. The ECCO client running on a web or application server connects to the EdgCapture payment server using TCP/IP. Depending upon the web and EdgCapture server configuration, this communication will be secured using the SSL TLSv1protocol with a certificate for the server signed by Edgil. SSL uses several different cryptographic processes to provide client and server authentication, encryption, and hashing. User authentication and database encryption are handled by the EdgCapture server.

Server authentication

The client is assured that it is connecting to the appropriate server for credit card transactions when it receives the server's certificate signed by Edgil's certificate authority. Edgil supplies the certificate to the EdgCapture system.

Message privacy

All communication between the client and the server is encrypted using 128-bit encryption. If third parties intercept an encrypted communication, they will not be able to decrypt the data stream without the client's private key.

Message validation

Using SSL, the client and server are both assured that the message they receive is the same as the message sent. SSL provides a cryptographic hashing algorithm that converts each message to a hash value that is transmitted along with the message. The recipient tests the message against the hash value to determine that the message has not been altered during transmission.

User authentication

Each client connection, whether initiated by an application or a human user, requires a logon and password. EdgCapture verifies that the user at the client machine is allowed to access the server and perform the current action when it receives the logon and password. Logons also restrict access to merchants and OEPs in EdgCapture itself.

Database security

All account numbers are encrypted in the EdgCapture database. Users see masked account numbers.

File security
All files containing credit information managed by Edgil are encrypted.

CardHolder Data Management
Referred to as CDM, Cardholder Data Management allows clients to store safe tokens instead of cardholder data. The tokens may be used in recurring or scheduled transactions in place of the cardholder data. A token can be created by sending a transaction request with cardholder data and getting a token in return, or by the CDM GetToken request. ECCO provides methods to manage the cardholder data using the token as a reference. See "Appendix E: Cardholder Data Management" on page 203 for a separate discussion of cardholder data management.

Note: CardHolder data Management is not currently supported in the C++ API implementation.

Transaction processing with ECCO

Processing of ECCO-based transactions generally follows the steps outlined in the previous section.

Successful transactions

The application uses ECCO to submit a transaction with all required information to EdgCapture. EdgCapture accepts the transaction request and enters the information into the database. The network service process (NSP) contacts the payment network requesting authorization, receives the authorization code, and records it in the database. EdgCapture then returns the transaction information, including the authorization code and status, in a reply message to the application. When the goods or services are shipped or delivered to the customer, the application sends EdgCapture a request to mark the transaction for capture. EdgCapture then moves the transaction to the marked for capture queue to await inclusion in the settlement batch. The transaction is now complete. After settlement the funds will be transferred from the customer's account to the merchant's.

Because credits do not need authorization, they are moved directly to the marked for capture queue upon receipt.

Unsuccessful transactions

ECCO is designed to handle situations in which a transaction is not successfully authorized or processing is disrupted. EdgCapture returns complete information, including processing status and appropriate error messages, with every response.

Declined authorization
If the authorization network does not approve the transaction, EdgCapture moves the transaction to the declined queue and returns a decline status to the application. The application can then request another card number, if it still has access to the customer. It can resubmit the transaction with the new card using the same connection.

Call center response
EdgCapture can either automatically decline all transactions receiving call center responses, or it can leave them in the held queue for later voice authorization. If EdgCapture is set to automatically decline these transactions, it returns a decline status to the application and moves the transaction to the declined queue. The application can then request another card number from the customer.

If the transaction is not automatically declined, an EdgCapture user must later call for voice authorization, enter the authorization code, and capture the transaction using the Transaction Manager. If the voice authorization is rejected, the EdgCapture user must decline the transaction and, if necessary, contact the customer.

Bad data
If the customer enters a credit card number that does not pass the algorithm check or has a bad expiration date, EdgCapture returns the appropriate error and moves the transaction to the error queue. The application can then request that the customer reenter the card number or date and resubmit the transaction with the correct data using the same connection. If the customer does not fix the data, the transaction remains in the error queue in EdgCapture. If the entire transaction was entered in error, the transaction may be voided, if the settlement period has not passed.

Communication problems or broken connections
The connection between the EdgCapture payment server and the client can be disrupted at any point during the communication of the transaction request. If the connection is disrupted or no connection can be established in the first place, no exchange between the ECCO client and EdgCapture takes place. In this case, the application can ask the customer to return later to complete the transaction.

If the authorization network is unavailable, EdgCapture returns this message to the application submitting a transaction. When this occurs, the application can request that the customer return later to complete the transaction. If the application submits transactions for automatic processing, EdgCapture will send the transaction for authorization when the network becomes available. In this case, the application must retrieve the transaction status later. If the authorization was declined, the customer should be contacted.

If the connection between the ECCO client and EdgCapture is broken after an authorization request is sent but before the reply is received, it is possible that the transaction has been successfully authorized. Depending upon its design, the application can retrieve the transaction status when the connection is reestablished, or it can report the error and request that the customer return later to complete the transaction.

If the connection is broken between authorization and marking for capture and EdgCapture has already returned the authorization code, the application can simply reconnect when possible and send the capture request. If the connection is broken after the capture request is sent, the application can send another request without affecting the transaction status.

Timeouts
An ECCO-based application can set a timeout period for each request. If the time needed for processing exceeds this value, the EdgCapture payment server reports an error. Timeout errors can occur when the payment network takes longer to process a transaction than the timeout value set by the application, for example. When the application receives a timeout error for an authorization request, it can treat it in the same way as an unavailable authorization network or a communications problem, as described in the previous section.

Duplicate transactions
If an application attempts to resubmit the same transaction for authorization, EdgCapture returns a duplicate transaction error and does not process the transaction again. The transaction data from the original request is also returned, including the status and the authorization code if the transaction was successfully authorized. In this case, the application can proceed with the next transaction or it can send a mark for capture request for the current one. Sending two mark for capture requests for the same authorized transaction does not result in a double charge.

The ECCO API

The ECCO API provides an interface to the EdgCapture payment server. Methods are supplied for establishing a secure connection to the EdgCapture machine, for submitting transactions for processing and receiving the results, and for checking transaction information. An application using the API can also pass all standard transaction data, including the 10 available user-defined data fields. Address Verification Service (AVS) is available, depending upon your payment processor.

The ECCO API is available in Java, PHP, .NET and C++.

See the associated “read me” file for each type of interface for system requirements and installation instructions.

Features

The ECCO API offers the following capabilities:

Payment processing

An application using the ECCO API can ensure that the transactions it submits to EdgCapture are processed correctly. That is, the customer will be charged for a purchase only once, even if a transaction is submitted repeatedly, and no extraneous transactions will appear on the customer’s statement if the transaction is cancelled.

Error handling

The ECCO API allows an application to easily reprocess transactions that contain bad data and or encounter errors. Transactions may be resubmitted using the same ECCOClient connection. Every request sent to the EdgCapture payment server returns a status code, and if unsuccessful, an error message. System messages are easily customized so that they can be reported directly to the customer.

Transaction identifiers

EdgCapture requires a unique identifier for each transaction within a merchant and OEP. The transaction ID prevents duplication of charges and allows an EdgCapture user to research the transaction.

Transaction information

The ECCO API includes methods to transmit and retrieve all available transaction data, including the 10 User Data fields.

Transaction Specific Data

The Ecco API includes a method to update transaction specific data associated with each transaction depending on its processing status.

Transaction Streaming

An application can use the same secure connection to send multiple requests, including multiple transaction requests, to the EdgCapture server. Each application instance has its own ECCOClient connection, and the server can handle multiple simultaneous connections.

Research

The ECCO API includes methods to retrieve the information for a single transaction, given the merchant, OEP, and transaction ID. If your site has purchased the directory option, an application can retrieve groups of transactions using any of the selection criteria available in EdgCapture's Transaction Manager. Users can also research groups of transactions submitted using ECCO through the Transaction Manager. ECCO-based transactions can be included in standard reports.

Debugging

ECCO has an extensive debugging facility, including logging from both the API and the client application that can be controlled by the application designer.

ECCO API Public Classes

The public classes provided by the ECCO API include:

- **ECCOClient** creates the secure connection to EdgCapture and sends transaction requests over the secure link.
- **MonetaryTransactionData** encapsulates the data for a monetary transaction request.
- **TransactionSelection** specifies criteria for requesting transaction information.
- **CardholderData** used for CDM requests such as requesting a token or updating the cardholder data referenced by a token.
- **ECCOStatusCodes** contains the status codes.
- **ECCOLog** is used to log when debugging an application.

To process transactions using the ECCO API, an application must:

1 Create an ECCOClient object.

2 Create a secure connection to EdgCapture using the Logon method of ECCOClient.

3 Create a MonetaryTransactionData object.

4 Set the data fields in the MonetaryTransactionData object.

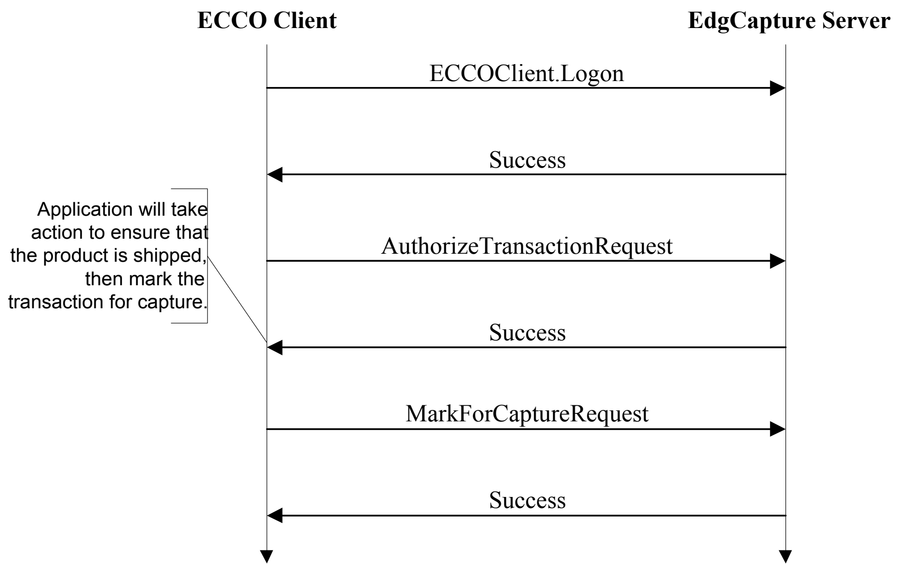
5 Call one of the ECCOClient's transaction request methods, passing the MonetaryTransactionData object as a parameter.

6 Process the results.

An application may send several requests to the server using the same connection by repeating steps 4–6.

To process a credit card sale, the application must send a request for authorization followed by a request to mark the transaction for capture. The message protocol illustrated here assumes that each step will be successful.

Message Protocol for Sale


ECCO Client
EdgCapture Server
ECCOClient.Logon
Success
Application will take action to ensure that the product is shipped, then mark the transaction for capture.
AuthorizeTransactionRequest
Success
MarkForCaptureRequest
Success

The ECCO Java API

This section describes the ECCO Java API. It includes short code samples illustrating basic functionality and complete class reference information.

Using the Java API

This section presents more detailed examples of how to process transactions and retrieve information from EdgCapture using the Java API. It also explains how to debug an application during development.

Setting up the ECCO.properties file

The ECCO API is delivered with a properties file that is set to use a “Fake” connection to EdgCapture. In “Fake” mode, messages are never actually transmitted to the EdgCapture server. All responses are generated by the java API itself.

When you are ready to submit a transaction to EdgCapture, you must edit the ECCO.properties file to change the connection type to “Ssl” and to specify the connection information for the EdgCapture server. Note that this parameter is case-sensitive and must be entered as documented here with an initial upper-case character.

See “Appendix D: Editing Properties Files” on page 199 for a complete description of the Properties file and the connection information for the EdgCapture server.

The ECCO.properties file also controls logging from the API. For information on logging, see “Logging during development” on page 25.

Processing a transaction

The procedure here covers a complete sale transaction, consisting of an authorization request and a mark for capture request. It also explains how to obtain a unique transaction identifier from EdgCapture for an application that does not produce its own identifiers.

Certification

The secure connection between the application and the EdgCapture server is established at startup of the application and before any transactions are submitted.

The ECCOClient method CertifyECCO currently uses edgilca.keystore as described in “Appendix D: Editing Properties Files” on page 199.

Create an ECCOClient object and call the ECCOClient method *certifyECCO* passing the required passwords and certificate alias to certify the client and server. Edgil recommends that the passwords be entered at startup by an actual user, instead of reading them from a configuration file, so that they are not stored as clear text anywhere on the system.

The first parameter for CertifyECCO is the file password for edgilca.keystore, the others are placeholders for future development.

```
ECCOClient myClient = new com.edgil.ecco.eccoapi.ECCOClient();
int status = myClient.certifyECCO(argv[0], argv[1], argv[2]);
```

The default values for edgilca.keystore can be used.

Argv[0]	ChangeIt	<case sensitive>	certificate pwd
Argv[1]	ChangeIt	<case sensitive>	certificate pwd
Argv[2]	edgilca	<case sensitive>	certificate alias

Creating the connection

In order to submit a transaction or an information request to EdgCapture, the application must establish a connection to the server by creating an ECCO client. When you create the ECCOClient object, you can also set up client-side logging for debugging purposes, as described in "Logging during development" on page 25.

Create the ECCOClient object.

```
ECCOClient myClient = new com.edgil.ecco.eccoapi.ECCOClient();
```

Call the ECCOClient method *logon,* passing logon and password as parameters for manual processing. Note that the logon and password belong to the application and can be used for any connection. The logon method opens a socket, certifying client and server, and sends the logon message to EdgCapture.

```
// Log onto the EdgCapture Server specifying manual processing
int status = myClient.logon("myLogon", "myPassword");
```

Test the results of the request. There are a few reasons for failure, including invalid logon or password, security errors, or connection problems.

Note: Edgil will provide the credentials for ECCOClient method logon to the EdgCapture server.

Default values are:

```
int status = myClient.logon("EccoClient", "EccoClient1");
```

Creating and populating the monetary transaction data

Create a *MonetaryTransactionData* object.

```
MonetaryTransactionData myTransactionData =
   new MonetaryTransactionData();
```

Set the data fields in the *MonetaryTransactionData* object. MonetaryTransactionData provides a set method for each individual data field, as well as methods to set several fields at once.

EdgCapture requires a unique transaction identifier for each transaction within a Merchant and OEP. The required fields for an authorization or credit request include:

Data	***Method***	***Description***
MerchantId	setMerchantId	Merchant identifier
OEPId	setOEPId	OEP identifier
Transaction Id	setTransactionId	Unique transaction identifier
Amount	setAmount	Amount of the sale or credit in format *nnnn.nn*; must be positive for all transaction types.
Payment information	setCreditCard setCheckDraft setDirectDebit setToken	Required credit card data includes AccountNumber and ExpirationDate. Required check draft and direct debit data includes ABANumber, AccountNumber, and PaymentDesignator. Token requires a valid CDM token
ECI	setECI	Electronic commerce indicator that describes the source of the transaction. For web applications, the value is generally 6, indicating that the customer's account data was transmitted using SSL.

See "Appendix C: Monetary Transaction Data" on page 195 for a complete description of the data fields. All data fields should be specified as strings.

If you are using a token for a monetary transaction, then the CardHolder Data fields are ignored..See "Appendix E: Cardholder Data Management" on page 203 for discussion of CDM and use of tokens.

The following example assumes that the application has created its own transaction id. It sets some data fields individually and others in logical groups:

Note: Actual setMerchantId and setOEPId values are determined by Edgil Associates.

```
myTransactionData.setMerchantId("0");
myTransactionData.setOEPId("A");
myTransactionData.setTransactionId("mN23d88");
myTransactionData.setAmount("95.40");
myTransactionData.setAccountNumber("9999999999999999");
myTransactionData.setExpirationDate("10","12");
myTransactionData.setECI("6");
myTransactionData.setFraudSecurityValue("123");
myTransactionData.setAddress("99 Wayward Lane","Boston", "MA",
   "01776");
myTransactionData.setName("Jane","Smith");
myTransactionData.setUserData1("Classified");
myTransactionData.setUserData2("auto");
myTransactionData.setUserData3("daily morning,weekly shopper");
myTransactionData.setUserData4("by Alice");
myTransactionData.setUserData6("Jan 5 2010");
```

Submitting the transaction to EdgCapture

Call the ECCOClient method *requestAuthorization* passing the previously created MonetaryTransactionData as a parameter.

```
int status = myClient.requestAuthorization(myTransactionData);
```

Processing the results

Test the return status from *requestAuthorization* to determine whether the transaction request has succeeded. The MonetaryTransactionData object will have all the information returned from EdgCapture, including the authorization code and other processing information for a successful transaction. If the request fails, the fields in the MonetaryTransactionData object remain as they were set before sending the request. If EdgCapture has recorded the transaction, it returns the transaction status, the current queue and other information.

SUCCESS

On ECCOStatusCodes.SUCCESS, the application should record the results, including MerchantId, OEPId, and TransactionId. For a successfully authorized transaction, the application must mark the transaction for capture to notify EdgCapture that the transaction is complete from the application side. If no mark for capture request is sent, the transaction will not be settled. See “Sending a mark for capture request” on page 19 for a description of this process.

To retrieve the authorization code, the current EdgCapture queue, or previously set transaction data, use the MonetaryTransactionData's get methods. For example:

```
string myAuthorizationCode =
   myTransactionData.getAuthorizationCode()
```

If using CardHolder Data Management, when a transaction result is returned use GetToken to retrieve the token that represents the cardholder data used for the transaction. This token may be stored for future transactions on that cardholder data. If a token was passed with the transaction request, the same token is returned. If no token was sent, a new or already existing token is returned related to the transaction's cardholder data.

FAILURE

Check the return status against the codes in ECCOStatusCodes. See “Appendix B: ECCO Status Codes” on page 189 for more information.

The type of failure determines the action needed. For data entry errors and declined credit cards, the customer may be asked to reenter information or another account number. The transaction can then be resubmitted using the same connection and Transaction Id.

```
switch status
   {
   case COMPROBLEM:
      //tell the customer to come back later
      break;
   case INVALIDACCOUNTNUMBER:
      //ask the customer to check it or provide another account
      break;
   }
```

If you want to display the message text, use the ECCOClient method *getStatusMessage* to retrieve it.

Sending a mark for capture request

The current MonetaryTransaction object can also be used to send the request to mark the transaction for capture.

After the application has ensured that the product has been shipped or delivered to the customer, the successfully authorized transaction must be marked for capture. If the MonetaryTransactionData object reporting the successful authorization still exists, it contains all the necessary information, and the application can use it for the request by calling the ECCOClient's *requestMarkForCapture* method.

```
int status = myClient.requestMarkForCapture(myTransactionData);
```

If the application must send the mark for capture message at a later time, follow the steps in the previous section to create a new *ECCOClient* connection and a new *MonetaryTransactionData* object. Set the MerchantId, OEPId, and TransactionId to the values stored for this transaction, and call *requestMarkForCapture*, passing the MonetaryTransactionData as a parameter.

```
   myTransactionData.setMerchantId("0");
   myTransactionData.setOEPId("A");
   myTransactionData.setTransactionId("mN23d88");

int status = myClient.requestMarkForCapture(myTransactionData);
```

Test the return status for success. See "Appendix B: ECCO Status Codes" on page 189 for information.

Reusing the connection for a subsequent transaction

You can reuse the same client connection for a new transaction or to resubmit a transaction after correcting errors in the data.

When the next transaction is entirely different from the current transaction, clear the MonetaryTransaction object of existing data using *clearAllFields*, which returns all fields to null.

```
myTransactionData.clearAllFields();
//now set the new ones
```

When a transaction request needs to be resent because of a minor error or when a new transaction is to be performed for the same customer, you do not need to clear the existing data before resetting the incorrect fields. In this example, the customer has entered a new account number:

```
myTransactionData.setAccountNumber("4444444444444");
```

To continue, submit the transaction request to EdgCapture.

Cleaning up

After all messages have been sent using the current ECCOClient, clean up and disconnect by calling the ECCOClient method *logoff*.

Retrieving transaction data for a single transaction

Each transaction is uniquely identified in the EdgCapture database by its Transaction Key, made up of the Merchant Id, the OEP Id, and the Transaction Id. To retrieve the data for any transaction, the application must have this information available. Note that you must perform a transaction selection in order to void a previously submitted transaction.

If no connection exists, create an ECCOClient and log on, as described in the previous section. Otherwise, use the existing ECCOClient connection.

Create the*TransactionSelection* object.

```
TransactionSelection mySelection = new TransactionSelection();
```

Use the TransactionSelection's *setSelectionTransactionKey* method to set the required Merchant Id, OEP Id and TransactionId that uniquely identify the transaction. You can also set the fields individually.

```
mySelection.setSelectionTransactionKey("0","B","BE345");
```

Call the ECCOClient method *requestDirectory* passing the previously created TransactionSelection as a parameter.

```
int status = myClient.requestDirectory(mySelection);
```

Test the return status for success. There are few reasons to fail, although it is possible if the application requests a Merchant or OEP to which it does not have access or if the requested transaction is not in the database.

If the request is successful, access the returned MonetaryTransactionData object.

```
if (status = ECCOStatusCodes.SUCCESS)
{
   //pick up the data
   MonetaryTransactionData myTransactionData;
   myTransactionData = mySelection.getNextTransaction();
}
```

Use MonetaryTransactionData's get methods to access the data. In this example, the application checks the queue to determine whether the transaction has been authorized.

```
string myQueue = myTransactionData.getQueueName();
```

To retrieve the data for another transaction using the same ECCOClient connection, reset the selection criteria and call *requestDirectory* again.

To close the connection, call the ECCOClient's *logoff* method.

Voiding a transaction

A void transaction nullifies a previous sale or credit transaction that has not yet been captured. A void can only be entered on the same day during the same settlement period as the transaction it is meant to offset. Voids are used to correct a mistake, such as an incorrect amount or card number, or to cancel a transaction submitted in error.

To void a transaction, request the transaction information as described in the previous section. Using the returned MonetaryTransactionData object, send the void request.

```
int status = myClient.requestVoid(myTransactionData);
```

Retrieving a directory of transactions

To retreive a group of transactions:

Create the *TransactionSelection* object.

```
TransactionSelection mySelection = new TransactionSelection();
```

Specify the number of transactions to return by setting a row count. Each row corresponds to the data for a single transaction. This example requests 20 transactions:

```
mySelection.setNumberRowsRequested(20);
```

Use TransactionSelection's set methods to specify the selection criteria. The selection criteria are defined in the ECCOClient. This example requests all Visa transactions from yesterday that are in the captured queue, that is, all Visa transactions that were included in the previous night's settlement batch.

```
mySelection.setSelectionPaymentType(ECCOClient.PAY_VISA);
mySelection.setSelectionQueue(ECCOClient.QUEUE_CAPTURED);
mySelection.setSelectionDateOptions(ECCOClient.DATE_OP_YESTERDAY);
```

Send the request to EdgCapture using the ECCOClient's *requestDirectory* method.

```
int status = myClient.requestDirectory(mySelection);
```

If the request is successful, access the MonetaryTransactionData objects. Use *getNumberRowsReturned* to determine how many transactions were returned. Note that the maximum number of rows for the current example is 20, as specified previously. To determine how many rows the query actually retrieved from the EdgCapture database, use TransactionSelection's *getNumberRowsFound* method.

```
if (status = ECCOStatusCodes.SUCCESS)
{
   int numberOfRows;
   MonetaryTransactionData myTransactionData;

   numberOfRows = mySelection.getNumberRowsReturned();
   for (i = 0; i < numberOfRows; i++)
   {
      myTransactionData = mySelection.getNextTransaction();
   }
}
```

Use the MonetaryTransactionData's get methods to access the data from individual fields.

Updating User Data for Transactions

Transactions have user data associated with them. These fields are labelled User1 through User10, plus the notes field. These fields are used to store additional data in EdgCapture for reference by the system sending transactions. For example, a system may use a user field to link EdgCapture transactions to it's own order by storing the order id in a user field.

The update user data request allows the updating of all user data fields plus the notes field. The data is sent via a MonetaryTransactionData object. This object may be returned via a transaction selection, re-used from a previously submitted transaction, or generated from scratch.

requestUpdateUserData will overwrite the existing user data for ALL of the following fields in the associated MonetaryTransactionData object: UserData1 through UserData10 and Notes. All other fields will remain unchanged.

Each transaction is uniquely identified in the EdgCapture database by its TransactionKey, made up of the Merchant Id, the OEP Id, and the Transaction Id. To update the user data for any transaction, the application must have this information available so that the transaction to be updated can be uniquely identified.

If no connection exists, create an ECCOClient and log on, as described in the previous sections. Otherwise, use the existing ECCOClient connection.

You could use a previously used MonetaryTransactionData object or create and populate one with the appropriate data in the transaction key, user fields, and notes field. Use the MonetaryTransactionData's *setTransactionKey* method to set the required MerchantId, OEPId and TransactionId that uniquely identify the transaction. Alternatively, you can also set the fields individually using the setters for TransactionId, MerchantId, and OEPId:

```
myUpdate.setTransactionKey("0","B","BE345") ;
```

Many times you will be re-using a MonetaryTransactionData object that was returned from a transaction selection or used to process a transaction. In this case you only need to update the user fields and notes fields as needed.

```
int status = myMonetaryTransactionData.setUserData1(“data”) ;
```

Once the data is ready, call the ECCOClient method *requestUpdateUserData*:

```
int status = myClient.requestUpdateUserData(
myMonetaryTransactionData) ;
```

Test the return status for success. If the request is successful, then the specified transaction will have the user data and notes field updated according to what was set in the MonetaryTransactionData.

To close the connection, call the ECCOClient’s logoff method.

Cardholder Data Management CDM

Cardholder data can be managed via the CDM requests provided by the ECCOClient class. The user creates a CardholderData object, then uses that object to send CDM requests and get replies.

ECCOClient provides three requests for managing cardholder data. requestCreateToken accepts cardholder data, and returns a token that can then be stored and used to reference the cardholder data. requestGetCardholderData accepts a token and returns the related cardholder data. requestUpdateCardholderData will update the data referenced by a token with new data sent in the request. For example, requestUpdateCardholderData may be used to change the expiration date on an account.

The rest of this section contains a general description of each of the CDM methods available. For detailed syntax, return values, and parameters, see the sections below related to your implementation language.

CDM requests use the CardholderData object. Below is an example of the population of a CardholderData object.

```
data.setAccountNumber( "4012888888881881" ) ;
data.setAddress ( "12 Pine Street", "Helton", "NC", "12345" ) ;
data.setExpirationDate ( "10", "12" ) ;
data.setName ( "Bebe", "Bekila" ) ;
data.setTelephone ( "222-222-2222" ) ;
```

Creating Tokens

requestCreateToken is a request that allows the client to store cardholder data in the Edgcapture data base for later use to process transactions. CardholderData objects are sent, and tokens are returned.

The requestCreateToken method is called through the ECCOClient object.

```
int status ;
 status = client.requestCreateToken ( data, false ) ;
```

Upon return, getToken can then be called to retrieve the created token. If the cardholder data already exists, then a previously existing token will be returned that matches the data, and the return status will be CARDHOLDER_DATA_ALREADY_EXISTS. The token related to that cardholder data will be returned. If new cardholder data is created the status will be SUCCESS.

Retrieving Cardholder Data

requestGetCardHolderData will return the cardholder data for a token. The token must be set in the data.

```
data.setToken (“123456789”) ;
```

Then the request can be made.

```
int status ;
status = client.requestGetCardholderData (data)
```

If the token is not found, the return status will be INVALID_TOKEN. Otherwise, the return status will be SUCCESS or some other error.

Updating Cardholder Data

CardHolder Data Management allows the client to update existing cardholder data by sending a token and updated data. The account number can not be updated. To insert a new account number you must use the requestCreateToken method.

After the update, the token will refer to the updated cardholder data. The previous data will remain in the database since it may be linked with transactions. When retrieving a directory of transactions using a token using the setSelectionToken method, the returned transactions will include those linked to the current cardholder data for that token, and all previous versions, since they all relate to the same account number.

When the update cardholder data request is processed, all data fields in the existing cardholder data are overwritten with the new data. Therefore, the client will typically read the existing data, change desired fields, and submit the cardholder data for update. The following example shows this procedure.

Read the data for the token.

```
data.clearAllFields() ;
data.setToken ( tokenForValidCD ) ;
if ( ( status = client.requestGetCardholderData ( data ) ) !=
   ECCOStatusCodes.SUCCESS )
   {error processing}
```

Modify desired fields.

```
data.setAddress ( "Updated Address", "Updated City", "RI", "55555"
   ) ;
```

Then request the update.

```
 if ( ( status = client.requestUpdateCardholderData ( data, false )
   ) !=
   ECCOStatusCodes.SUCCESS )
   {error processing}
```

Logging during development

The ECCO API provides logging from both the EdgCapture side and ECCO API side. Logging on EdgCapture is a normal part of operation. Log records may be viewed from the Flight Recorder and Log Viewer, which is part of the EdgCapture user interface. Logging from the ECCO API is offered as a tool for debugging during the development of an application only and should not be used during normal operation. The ECCO API also provides an ECCOLog class which may be used directly by the application to log its own messages.

Logging from the API

The ECCO API uses the SimpleSocketServer task to log to a user-specified file, which contains a single day's messages. The file is truncated daily at the first message received after midnight, and the previous day's messages are saved in a file with a date extension. If desired, ECCO logging can also be directed to the console in addition to the file, or to the console only.

API logging is controlled by an ECCO properties file, whose options may be modified by method calls within the application. The ECCO.properties file is delivered with logging set to report only fatal errors to the console; this is the default for normal operation. To set up logging for development, you can either edit the default ECCO.properties file or create another properties file that specifies logging by uncommenting the logging sections for the output you want. See "Appendix D: Editing Properties Files" on page 199 for information on editing the properties file. Note that if you use the default ECCO.properties file for development, you must edit it after the debugging process is finished to prevent API logging in a production environment.

ECCO logging uses three priorities: INFO, WARN, and ERROR. All public API methods will be logged as INFO. All exceptions and any other errors will be logged as ERROR. If the priority level is set to INFO, all priority levels are logged. The default priority level is specified in the ECCO properties file, but the application may reset the priority as desired.

Logging from the application

The ECCO API provides the ECCOLog class both for logging by the API and for direct logging from the client application. Logging from the application requires code modifications, but it has the advantage of turning logging on and off at desired locations, making the log file easier to read.

The ECCOLog class provides three same three static priorities for use by the ECCO classes and an application using the ECCO Client: INFO, WARN and ERROR. Because all method calls from the API are logged as INFO, you might want to set the priority of method calls from the application to WARN or ERROR. You can then limit the size of the log by changing the priority in the properties file to WARN, which will log only WARN and ERROR levels.

In order to log, the application must create an ECCOClient that specifies logging in its constructor and then create the ECCOLog object. Generally the ECCOClient will be the object created for certification of the connection between the client and EdgCapture. Two parameters must be provided when constructing the ECCOClient for logging:

- The thread- or task-specific session identifier. Servlets can use their HTTP session id; tasks can use their name, if it is unique. This string is included in the logging entry to identify entries from the same source.
- The pathname of the ECCO properties file that specifies logging, either the default ECCO.properties file or another file.

The following sample log entries use “TestingFromIDE” is the first parameter:

```
2001-07-23 12:57:50,057 [main] INFO com.edgil.ecco.eccoapi.
   MonetaryTransactionData TestingFromIDE - constructor
2001-07-23 12:57:50,077 [main] INFO com.edgil.ecco.eccoapi.
   MonetaryTransactionData TestingFromIDE - constructor
2001-07-23 12:57:50,077 [main] INFO com.edgil.ecco.eccoapi.
   MonetaryTransactionData TestingFromIDE - setTransac-
   tionId(MyUniqueID2001)
2001-07-23 12:57:50,087 [main] INFO com.edgil.ecco.eccoapi.
   MonetaryTransaction TestingFromIDE - setCredit-
   Card(******,12,04)
2001-07-23 12:57:50,117 [main] INFO com.edgil.ecco.eccoapi.
   MonetaryTransactionData TestingFromIDE - setMerchantId(0)
```

After creating the ECCOClient with session identifier and properties file, the application must create its own ECCOLog object with the name of the caller doing the logging, which will appear in the log entry. This example logs the informational message “we’re here” when MyApplicationClass is called:

```
ECCOClient theClient = new com.edgil.ecco.eccoapi.ECCOClient
   ("mySession", "c:\edgil\data\myPropertiesFile.properties");

ECCOLog logMyApplicationClass =
   new ECCOLog(MyApplicationClass.class.getName());
logMyApplicationClass.setPriority(ECCOLog.WARN)
logMyApplicationClass.warn("we're here");
```

Setting up the SimpleSocketServer for logging

ECCO API logging uses the SimpleSocketServer task, which receives messages from any user of the ECCO API, such as a background application or a web server's JVM. Edgil provides a default SimpleSocketServerEcco.properties file that logs all messages both to the console and to a specified destination file. The default file contains a Windows-specific pathname for the log file. Users of non-Windows operating systems must edit the properties file to specify an appropriate pathname for their environment. Note that the directory for the log file must exist, or logging will fail. For information on the SimpleSocketServerEcco.properties file, see "Appendix D: Editing Properties Files" on page 199.

Included in the ECCO API installation is the StartSimpleSocketServer.bat file to start the server on Windows. You can also start the SimpleSocketServer by passing it a port number and the path/file name for its properties file:

```
java java.org.apache.log4j.net.SimpleSocketServer 3421
   /ecco/data/SimpleSocketServerECCO.properties
```

Note that the port number must match the port specified for the SocketApp appender in the ECCO.properties file:

```
   log4j.appender.SocketApp.Port=3421
```

ECCO Java API Class Reference

This section contains descriptions of the ECCO java API classes contained in the package com.edgil.ecco.eccoapi.

Package
com.edgil.ecco.eccoapi

Class Summary	
Classes	
`ECCOClient`	An ECCOClient object encapsulates a connection to the EdgCapture server.
`ECCOLog`	ECCOLog provides optional logging for the ECCO API and the users of the ECCO API.
`ECCOStatusCodes`	This class contains the ECCO Status Codes as static final ints.
`MonetaryTransactionData`	A MonetaryTransactionData object contains the data sent to or returned from EdgCapture for a Monetary Transaction in the EdgCapture database.
`TransactionSelection`	A TransactionSelection object encapsulates the transaction selection criteria for retrieving transactions from the EdgCapture database.
`CardHolderData`	A CardHolderData object includes the CDM requests such as requesting a token or updating the cardholder data referenced by a token.

com.edgil.ecco.eccoapi ECCOClient

Declaration

```
public class ECCOClient

java.lang.Object
  |
  +--com.edgil.ecco.eccoapi.ECCOClient
```

Description

An ECCOClient object encapsulates a connection to the EdgCapture server. Through the ECCOClient object, the user logs onto the EdgCapture server, sends requests, and receives replies.

Member Summary	
Fields	
`public static final`	`CHECK_DESIGNATOR_CONSUMER` Payment Designator for Checks
`public static final`	`CHECK_DESIGNATOR_CORPORATE` Payment Designator for Checks
`public static final`	`DATE_OP_DATERANGE` Directory Date Option
`public static final`	`DATE_OP_LASTWEEK` Directory Date Option
`public static final`	`DATE_OP_THISWEEK` Directory Date Option
`public static final`	`DATE_OP_TODAY` Directory Date Option
`public static final`	`DATE_OP_YESTERDAY` Directory Date Option
`public static final`	`FRAUD_SECURITY_RESPONSE_INVALID` Fraud Security Response
`public static final`	`FRAUD_SECURITY_RESPONSE_MATCH` Fraud Security Response
`public static final`	`FRAUD_SECURITY_RESPONSE_NA` Fraud Security Response
`public static final`	`FRAUD_SECURITY_RESPONSE_NO_DATA` Fraud Security Response
`public static final`	`FRAUD_SECURITY_RESPONSE_NO_MATCH` Fraud Security Response
`public static final`	`FRAUD_SECURITY_RESPONSE_NOT_ON_CARD` Fraud Security Response
`public static final`	`FRAUD_SECURITY_RESPONSE_NOT_PROCESSED` Fraud Security Response
`public static final`	`FRAUD_SECURITY_RESPONSE_SHOULD_HAVE` Fraud Security Response
`public static final`	`FRAUD_SECURITY_RESPONSE_UNSUPPORTED` Fraud Security Response

Member Summary	
`public static final`	`PAY_AMERICAN_EXPRESS` Payment Type
`public static final`	`PAY_CARTE_BLANCHE` Payment Type
`public static final`	`PAY_CASH` Payment Type
`public static final`	`PAY_CHECK_DRAFT` Payment Type
`public static final`	`PAY_CREDIT_CARD` Payment Type
`public static final`	`PAY_DINERS_CLUB` Payment Type
`public static final`	`PAY_DIRECT_DEBIT` Payment Type
`public static final`	`PAY_DISCOVER` Payment Type
`public static final`	`PAY_MASTER_CARD` Payment Type
`public static final`	`PAY_PAPER_CHECK` Payment Type
`public static final`	`PAY_VISA` Payment Type
`public static final`	`QUEUE_AUTHORIZED` Queue Name
`public static final`	`QUEUE_CAPTURED` Queue Name
`public static final`	`QUEUE_DECLINED` Queue Name
`public static final`	`QUEUE_ERROR` Queue Name
`public static final`	`QUEUE_HELD` Queue Name
`public static final`	`QUEUE_INPUT` Queue Name
`public static final`	`QUEUE_MARKED_FOR_CAPTURE` Queue Name
`public static final`	`QUEUE_RECORDED_ONLY` Queue Name
`public static final`	`QUEUE_UNKNOWN_DISPOSITION` Queue Name
`public static final`	`QUEUE_VOIDED` Queue Name
`public static final`	`TRAN_AUTHORIZE` Transaction Type
`public static final`	`TRAN_CAPTURE_HELD_TRANSACTION` Transaction Type
`public static final`	`TRAN_CREDIT` Transaction Type

Member Summary	
`public static final`	`TRAN_DECLINE_HELD_TRANSACTION` Transaction Type
`public static final`	`TRAN_MARKED_FOR_CAPTURE` Transaction Type
`public static final`	`TRAN_RECORD_ONLY` Transaction Type
`public static final`	`TRAN_VOID` Transaction Type
Constructors	
`public`	`ECCOClient()` Constructs an ECCO Client that does not log and uses the default properties file.
`public`	`ECCOClient(String)` Constructs an ECCO Client that does not log and uses a user-specified properties file.
`public`	`ECCOClient(String, String)` Constructs an ECCO Client that logs and uses a user-specified properties file.
Methods	
`public static int`	`certifyECCO(char[], char[], char[])` Certifies ECCO for a secure connection.
`public static int`	`certifyECCO(char[], char[], char[], String)` Certifies ECCO for a secure connection specifying logging identifier.
`public String`	`getStatusMessage()` Gets the Status message from Logon or the last request sent to EdgCapture.
`public void`	`logoff()` Closes the Connection to the ECCO Server
`public int`	`logon(String, String)` Logs on to the ECCO Server with a default setting of Manual for Monetary Transactions
`public int`	`logon(String, String, boolean)` Logs on to the ECCO Server with a Client specified setting of Manual or Automatic for Manual Transactions.
`public int`	`logon(String, String, String)` Logs on to the ECCO Server with a default setting of Manual for Monetary Transactions and a client specified process source.
`public int`	`logon(String, String, String, boolean)` Logs on to the ECCO Server with a Client specified setting of Manual or Automatic for Manual Transactions and a client specified process source.
`public int`	`requestAuthorization(MonetaryTransactionData)` Sends request for authorization of sale transaction to EdgCapture.
`public int`	`requestCredit(MonetaryTransactionData)` Sends request to enter credit transaction to EdgCapture.
`public int`	`requestDirectory(TransactionSelection)` Obtains transaction data from EdgCapture.
`public int`	`requestMarkForCapture(MonetaryTransactionData)` Sends request to mark an authorized sale for capture.

Member Summary	
public int	`requestRecordOnly(MonetaryTransactionData)` Sends request to enter a record only transaction to EdgCapture.
public int	`requestUniqueId(MonetaryTransactionData)DEPRECATED` Obtains a transactionId from EdgCapture.
public int	`requestVoid(MonetaryTransactionData)` Sends a request to void a transaction in the EdgCapture database that is not yet captured.
public int	`requestUpdateUserData(MonetaryTransactionData)` Updates all UserData fields and the Notes field for the specified transaction.
public int	`requestCreateToken(CardHolderData, AddressValidation )` Creates cardholder data and returns related token
public int	`requestGetCardholderData ( CardholderData )` Returns cardholder data associated with the specified token.
public void	`setECCOPriority(int)` Resets the priority for logging by all of com.edgil.ecco.eccoapi

Inherited Member Summary
Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Fields

CHECK_DESIGNATOR_CONSUMER

`public static final String` **`CHECK_DESIGNATOR_CONSUMER`**

Payment designator for checks: consumer checking account

CHECK_DESIGNATOR_CORPORATE

`public static final String` **`CHECK_DESIGNATOR_CORPORATE`**

Payment designator for checks: corporate checking account

DATE_OP_DATERANGE

`public static final int` **`DATE_OP_DATERANGE`**

Date option for transaction selection: transactions between specified start and end date

DATE_OP_LASTWEEK

`public static final int` **`DATE_OP_LASTWEEK`**

Date option for transaction selection: transactions from previous week

DATE_OP_THISWEEK

`public static final int` **`DATE_OP_THISWEEK`**

Date option for transaction selection: transactions from current week

DATE_OP_TODAY

`public static final int` **`DATE_OP_TODAY`**

Date option for transaction selection: today's transactions

DATE_OP_YESTERDAY

`public static final int` **`DATE_OP_YESTERDAY`**

Date option for transaction selection: yesterday's transactions

FRAUD_SECURITY_RESPONSE_INVALID

`public static final int` **`FRAUD_SECURITY_RESPONSE_INVALID`**

Fraud security response: fraud security value is invalid; for example, incorrect number of characters

FRAUD_SECURITY_RESPONSE_MATCH

`public static final int` **`FRAUD_SECURITY_RESPONSE_MATCH`**

Fraud security response: fraud security value matches account

FRAUD_SECURITY_RESPONSE_NA

`public static final int` **`FRAUD_SECURITY_RESPONSE_NA`**

Fraud security response: fraud security value not applicable; for example, credits, checks, other transactions not sent for authorization

FRAUD_SECURITY_RESPONSE_NO_DATA

`public static final int` **`FRAUD_SECURITY_RESPONSE_NO_DATA`**

Fraud security response: no fraud security value submitted or no response from network for other reasons

FRAUD_SECURITY_RESPONSE_NO_MATCH

`public static final int` **`FRAUD_SECURITY_RESPONSE_NO_MATCH`**

Fraud security response: fraud security value does not match account

FRAUD_SECURITY_RESPONSE_NOT_ON_CARD

`public static final int` **`FRAUD_SECURITY_RESPONSE_NOT_ON_CARD`**

Fraud security response: merchant indicated no fraud security value on card

FRAUD_SECURITY_RESPONSE_NOT_PROCESSED

`public static final int` **`FRAUD_SECURITY_RESPONSE_NOT_PROCESSED`**

Fraud security response: network failed to process fraud security value

FRAUD_SECURITY_RESPONSE_SHOULD_HAVE

`public static final int` **`FRAUD_SECURITY_RESPONSE_SHOULD_HAVE`**

Fraud security response: merchant indicated fraud security value is not on card, but network claims it should be

FRAUD_SECURITY_RESPONSE_UNSUPPORTED

`public static final int` **`FRAUD_SECURITY_RESPONSE_UNSUPPORTED`**

Fraud security response: fraud security program not supported by issuing bank

PAY_AMERICAN_EXPRESS

`public static final String` **`PAY_AMERICAN_EXPRESS`**

Payment type

PAY_CARTE_BLANCHE

`public static final String` **`PAY_CARTE_BLANCHE`**

Payment type

PAY_CASH

`public static final String` **`PAY_CASH`**

Payment type; not currently implemented

PAY_CHECK_DRAFT

`public static final String` **`PAY_CHECK_DRAFT`**

Payment type

PAY_CREDIT_CARD

`public static final String` **`PAY_CREDIT_CARD`**

Payment type

PAY_DINERS_CLUB

`public static final String` **`PAY_DINERS_CLUB`**

Payment type

PAY_DIRECT_DEBIT

```
public static final String PAY_DIRECT_DEBIT
```

Payment type

PAY_DISCOVER

```
public static final String PAY_DISCOVER
```

Payment type

PAY_MASTER_CARD

```
public static final String PAY_MASTER_CARD
```

Payment type

PAY_PAPER_CHECK

```
public static final String PAY_PAPER_CHECK
```

Payment type

PAY_VISA

```
public static final String PAY_VISA
```

Payment type

QUEUE_AUTHORIZED

```
public static final String QUEUE_AUTHORIZED
```

Queue containing successfully authorized transactions

QUEUE_CAPTURED

```
public static final String QUEUE_CAPTURED
```

Queue containing transactions submitted for settlement

QUEUE_DECLINED

```
public static final String QUEUE_DECLINED
```

Queue containing transactions rejected by network

QUEUE_ERROR

```
public static final String QUEUE_ERROR
```

Queue containing transactions with data or processing errors

QUEUE_HELD

`public static final String` **`QUEUE_HELD`**

Queue containing transactions that received call center response from network and need manual intervention

QUEUE_INPUT

`public static final String` **`QUEUE_INPUT`**

Queue containing transactions that have been entered, but not yet processed

QUEUE_MARKED_FOR_CAPTURE

`public static final String` **`QUEUE_MARKED_FOR_CAPTURE`**

Queue containing transactions that have been authorized and marked for capture (ready to be included in settlement)

QUEUE_RECORDED_ONLY

`public static final String` **`QUEUE_RECORDED_ONLY`**

Queue containing transactions that have been entered for recording purposes

QUEUE_UNKNOWN_DISPOSITION

`public static final String` **`QUEUE_UNKNOWN_DISPOSITION`**

Queue containing transactions with unknown errors

QUEUE_VOIDED

`public static final String` **`QUEUE_VOIDED`**

Queue containing voided transactions

TRAN_AUTHORIZE

`public static final String` **`TRAN_AUTHORIZE`**

Transaction type: submit the amount for authorization

TRAN_CAPTURE_HELD_TRANSACTION

`public static final String` **`TRAN_CAPTURE_HELD_TRANSACTION`**

Transaction type: mark the transaction for capture after successful voice authorization

TRAN_CREDIT

`public static final String` **`TRAN_CREDIT`**

Transaction type: submit the amount to be credited to the account

TRAN_DECLINE_HELD_TRANSACTION

```
public static final String TRAN_DECLINE_HELD_TRANSACTION
```

Transaction type: decline the transaction after rejected voice authorization

TRAN_MARKED_FOR_CAPTURE

```
public static final String TRAN_MARKED_FOR_CAPTURE
```

Transaction type: mark the transaction to be included in settlement

TRAN_RECORD_ONLY

```
public static final String TRAN_RECORD_ONLY
```

Transaction type: enter the transaction data without submitting it to the payment network

TRAN_VOID

```
public static final String TRAN_VOID
```

Transaction type: reverse a specified authorize or credit transaction

Constructors

ECCOClient()

```
public ECCOClient()
```

Constructs an ECCO Client that does not log and uses the default properties file. The default ECCO properties file is ECCO.properties.

ECCOClient(String)

```
public ECCOClient(String eccoPropertiesFile)
```

Constructs an ECCO Client that does not log and uses a user-specified properties file.

Parameters:

`eccoPropertiesFile` - The path and file name for the ECCO properties file. This parameter overrides the default ECCO.properties file.

ECCOClient(String, String)

```
public ECCOClient(String sessionLoggingID,
                  String eccoPropertiesFile)
```

Constructs an ECCO Client that logs and uses a user-specified properties file.

Parameters:

`sessionLoggingID` - A unique id to tie together log messages from a single application or thread.

`eccoPropertiesFile` - The path and file name for the ECCO properties file; may be either ECCO.properties or a user-specified properties file, but it must be specified.

Methods

certifyECCO(char[], char[], char[])

```
public static int certifyECCO(char[] theKeyStorePass,
                 char[] theCertificatePass,
                 char[] theCertificateAlias)
```

Certifies ECCO for a secure connection. This is a static call that can be made once per application before logging on to the ECCO Server. The char arrays are zeroed out after use.

Parameters:

`theKeyStorePass` - The password for the keystore

`theCertificatePass` - The password for the local certificate

`theCertificateAlias` - The alias for the local certificate

Returns:

An ECCOStatusCode.

certifyECCO(char[], char[], char[], String)

```
public static int certifyECCO(char[] theKeyStorePass,
                 char[] theCertificatePass,
                 char[] theCertificateAlias
                 String sessionLoggingID)
```

Certify ECCO for a secure connection, logging any exceptions. This is a static call that must be made once for each application before logging on to the ECCOServer.

Parameters:

`theKeyStorePass` - The password for the keystore

`theCertificatePass` - The password for the local certificate

`theCertificateAlias` - The alias for the local certificate

`sessionLoggingID` - A unique id to tie together log messages from one or more ECCO objects.

Returns:

An ECCOStatusCode.

getStatusMessage()

```
public String getStatusMessage()
```

Gets the Status message from the last request sent to EdgCapture.

Returns:

The Status Message from the last request sent to the Server.

logoff()

```
public void logoff()
```

Closes the connection to the ECCO Server

logon(String, String)

```
public int logon(String logonID,
                 String password)
```

Logs on to the ECCO Server with a default setting of manual for monetary transactions

Parameters:

`logonID` - The client's user id for identification with the ECCO Server

`password` - The client's password for identification with the ECCO Server

Returns:

An ECCOStatusCode

logon(String, String, boolean)

```
public int logon(String logonID,
                 String password,
                 boolean automatic)
```

Logs on to the ECCO Server with a Client specified setting of manual or automatic for manual transactions.

Parameters:

`logonID` - The client's user id for identification with the ECCO Server

`password` - The client's password for identification with the ECCO Server

`automatic` - If TRUE, specify automatic for monetary transactions. IF FALSE, specify manual for monetary transactions.

Returns:

An ECCOStatusCode.

logon(String, String, String)

```
public int logon(String logonID,
                 String password,
                 String processSource)
```

Logs on to the ECCO Server with a default setting of manual for monetary transactions and a client specified process source. Process source is used to tie together transactions from one or more points of origin or one or more users. Then it may be used to group together transactions sharing a process source into a common directory selection.

Parameters:

`logonID` - The client's user id for identification with the ECCO Server

`password` - The client's password for identification with the ECCO Server

`processSource` - The client's process identity for consolidating multiple users or points of origin.

Returns:

An ECCOStatusCode.

logon(String, String, String, boolean)

```
public int logon(String logonID,
                 String password,
                 String processSource,
                 boolean automatic)
```

Logs on to the ECCO Server with a user-specified setting of manual or automatic for manual transactions and a client specified process source.

Parameters:

`logonID` - The client's user id for identification with the ECCO Server

`password` - The client's password for identification with the ECCO Server

`processSource` - The client's process identifier for consolidating multiple users within an asynchronous application.

`automatic` - TRUE specifies automatic for monetarytransactions; FALSE specifies manual for monetary transactions.

Returns:

An ECCOStatusCode.

requestAuthorization(MonetaryTransactionData)

```
public int requestAuthorization(MonetaryTransactionData theData)
```

Requests authorization for a transaction. Sends a request to EdgCapture to obtain authorization for the sale represented by the passed MonetaryTransactionData.

Parameters:

`theData` - MonetaryTransactionData object with the minimum of the following required information provided: merchantId, oepId, transactionId, amount, ECI, payment fields for credit card or check or direct debit.

Returns:

An ECCOStatusCode. If SUCCESS, the transaction will be authorized in the EdgCapture database. The MonetaryTransactionData object will contain the authorization code and date, the reference number and other EdgCapture information, which may be obtained by gets.

requestCredit(MonetaryTransactionData)

```
public int requestCredit(MonetaryTransactionData theData)
```

Submits a credit for capture. Sends a request to EdgCapture to process the credit represented by the passed MonetaryTransactionData. Successful credits are placed in the marked for capture queue.

Parameters:

`theData` - MonetaryTransactionData object with the minimum of the following required information provided: merchantId, oepId, transactionId, amount, account number, expiration month, expiration year, ECI.

Returns:

An ECCOStatusCode. If SUCCESS, the transaction will be marked for capture in the EdgCapture database. The MonetaryTransactionData object will contain all the EdgCapture information which may be obtained by gets.

requestDirectory(TransactionSelection)

`public int requestDirectory(TransactionSelection theDirectory)`

Obtains transaction data from EdgCapture. Sends a request to EdgCapture for data from a single specified transaction or for a directory of transactions.

Parameters:

`theDirectory` - TransactionSelection object with any desired selection limitation set.

Returns:

An ECCOStatusCode. If SUCCESS, the TransactionSelection object will contain the returned list of MonetaryTransactionData objects. Use the TransactionSelection's getNumberRowsReturned and getNextTransaction for access to the MonetaryTransactionData objects.

requestMarkForCapture(MonetaryTransactionData)

`public int requestMarkForCapture(MonetaryTransactionData theData)`

Marks an authorized sale for capture. Sends a request to EdgCapture to mark for capture the authorized sale represented by the passed MonetaryTransactionData.

Parameters:

`theData` - MonetaryTransactionData object with the minimum of the following required information provided: merchantId, oepId, transactionId

Returns:

An ECCOStatusCode. If SUCCESS, the transaction will be marked for capture in the EdgCapture database. The MonetaryTransactionData object will contain all the EdgCapture information which may be obtained by gets.

requestRecordOnly(MonetaryTransactionData)

`public int requestRecordOnly(MonetaryTransactionData theData)`

Submits transaction data for entry into the database without sending it to the payment network. Sends a request to EdgCapture to enter the transaction represented by the passed MonetaryTransactionData for recording purposes.

Parameters:

`theData` - MonetaryTransactionData object with the minimum of the following required information provided: merchantId, oepId, transactionId, amount, accountNumber. Note that expiration date information is not required.

Returns:

An ECCOStatusCode. If SUCCESS, the transaction will be entered in the EdgCapture database.

requestVoid(MonetaryTransactionData)

```
public int requestVoid(MonetaryTransactionData theData)
```

Voids a transaction in the EdgCapture database that is not yet captured. Sends a request to EdgCapture to void the transaction represented by the passed MonetaryTransactionData. Voids are only possible before the transaction has been captured and moved from the MarkedForCapture queue to the Captured queue. Because voids require all available data for the transaction to be voided, you must first retrieve the MonetaryTransactionData object from EdgCapture using the TransactionSelection requestDirectory method.

Parameters:

`theData` - MonetaryTransactionData object with all available data for the transaction to be voided.

Returns:

An ECCOStatusCode. If SUCCESS, the transaction will be moved from the Authorized or MarkedForCapture queue to the Voided queue. The MonetaryTransactionData object will contain all the EdgCapture information which may be obtained by gets.

requestUpdateUserData(MonetaryTransactionData)

```
public int requestUpdateUserData(MonetaryTransactionData theData)
```

Updates all UserData fields and the Notes field for the specified transaction. theData is usually an object previously used in another transaction or returned from a call to requestDirectory.

Parameters:

`theData` - MonetaryTransactionData object with the minimum of the following required information provided: merchantId, oepId, transactionId, UserData1 – UserData10, Notes

Returns:

An ECCOStatusCode. If ECCO_SUCCESS, the user data for the transaction has been updated.

requestCreateToken(CardholderData data, boolean AVSValidation

```
public int requestCreateToken(CardholderData data, boolean AVSValidation
              )
```

This request will create a set of cardholder data in the EdgCapture database. The cardholder data can be referenced using the returned token.

Upon sending the request, the cardholder data should be populated with valid cardholder data (see definition of the cardholder data object). The token field should be empty. Upon return, the token field is populated with the new token. If the passed cardholder data already exists, the existing token that references that data is returned.

If the AVSValidation parameter is true, the cardholder data is validated with respect to address validation, all address fields needed for address validation must be filled in.

Parameters:

`theData` - CardholderData

Boolean AVSValidation - if true then fields required for address validation must be filled in, minimally the zip code and last name..

Returns:

An ECCOStatusCode. If ECCO_SUCCESS, the cardholder data has been created in the EdgCapture database and the returned token can be used to refer to that data in subsequent monetary transactions or CDM requests.

requestGetCardholderData(CardholderData data)

```
public int requestGetCardholderData( CardholderData data)
```

Uses the token field in the cardholder data to retrieve the related cardholder data.

Parameters:

data – CardholderData object with the token filled in.

Returns:

An ECCOStatusCode. If SUCCESS, the CardholderData object will be filled in with the data related to the token that was passed.

requestUpdateCardholderData(CardholderData data, boolean AVSValidation)

```
public int requestUpdateCardholderData( CardholderData data,
              boolean AVSValidation )
```

Updates the cardholder data indicated by the token. The existing cardholder data will be completely replaced by the contents of the passed data object. Previous versions of the cardholder data are retained for searching and linking to pre-existing transactions. Validation is performed in the same way as the getToken request.

Parameters:

data – CardholderData object with the token filled in.

AVSValidation – if true, then data must contain fields needed for address validation, or the update will fail. Minimally this is zip code and last name.

Returns:

An ECCOStatusCode. If SUCCESS, the cardholder data was updated.

setECCOPriority(int)

```
public void setECCOPriority(int priority)
```

Resets the priority for logging by all of com.edgil.ecco.eccoapi

Parameters:

`priority` - One of the three logging priority levels: INFO, WARN, or ERROR.

com.edgil.ecco.eccoapi ECCOLog

Declaration

```
public class ECCOLog

java.lang.Object
  |
  +--com.edgil.ecco.eccoapi.ECCOLog
```

Description

ECCOLog provides optional logging for the ECCO API and the users of the ECCO API. Until an ECCOClient is created in the thread or application using the ECCOLog, logging calls will be ineffective. The user of ECCOLog may change the Priority of logging at any time to turn logging on or off.

Member Summary	
Fields	
`public static final`	`ERROR`
`public static final`	`INFO`
`public static final`	`WARN`
Constructors	
`public`	`ECCOLog(String)`
`public`	`ECCOLog(String, String)`
`public`	`ECCOLog(String, String, String)`
Methods	
`public void`	`error(String)`
`public void`	`info(String)`
`public boolean`	`setPriority(int)`
`public void`	`warn(String)`

Inherited Member Summary
Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Fields

ERROR

```
public static final int ERROR
```

INFO

```
public static final int INFO
```

WARN

```
public static final int WARN
```

Constructors

ECCOLog(String)

```
public ECCOLog(String className)
```

Sets logging for a specified class.Construction of an ECCOLog object always requires a class name, so that the log records are marked with their source.

Parameters:

`className` - Name of the class to be logged.

ECCOLog(String, String)

```
public ECCOLog(String className,
               String sessionLoggingID)
```

Sets logging for a specified class, using the specified session ID.

Parameters:

`className` - Name of the class to be logged.

`sessionLoggingID` - Identifier to tie messages from this thread together.

ECCOLog(String, String, String)

```
public ECCOLog(String className,
               String sessionLoggingID,
               String propertiesFile)
```

Sets logging for a specified class, using the specified session logging ID and the specified properties file.

Parameters:

`className` - Name of the class to be logged.

`sessionLoggingID` - Identifier to tie messages from this thread together.

`propertiesFile` - Pathname of the properties file.

Sets logging for a specified class, using the specified session logging ID, and the specified properties file.

Methods

error(String)

`public void error(String theLogMessage)`

Writes the passed message to the debugging log record if logging is active at this priority level.

Parameters:

`theLogMessage` - Message with priority level of error.

info(String)

`public void info(String theLogMessage)`

Writes the passed message to the debugging log record if logging is active at this priority level.

Parameters:

`theLogMessage` - Message with priority level of info.

setPriority(int)

`public void setPriority(int priority)`

Sets debugging logging priority to the specified level.

Parameters:

`iPriority` - One of the three logging priority levels: ECCOLOG_INFO, ECCOLOG_WARN, or ECCOLOG_ERROR.

warn(String)

`public void warn(String theLogMessage)`

Writes the passed message to the debugging log record if logging is active at this priority level.

com.edgil.ecco.eccoapi ECCOStatusCodes

Declaration

```
public class ECCOStatusCodes

java.lang.Object
  |
  +--com.edgil.ecco.eccoapi.ECCOStatusCodes
```

Description

This class contains the ECCO Status Codes as static final ints. The codes correspond to the keys in the ECCOStatusMessages properties file. For complete explanations, see "Appendix B: ECCO Status Codes" on page 185.

Member Summary	
Fields	
public static final	ABA_ROUTING_NUMBER_MISSING
public static final	ABA_ROUTING_NUMBER_NOT_ALLOWED
public static final	ADPAY_PAYMENT_DESIGNATOR_MISSING
public static final	ADPAY_PAYMENT_DESIGNATOR_NOT_ALLOWED
public static final	ALREADY_ACKNOWLEDGED
public static final	ALREADY_LOGGED_ON
public static final	AMOUNT_MISSING
public static final	AMOUNT_NOT_ALLOWED
public static final	AUTH_NOT_CAPTURED
public static final	AUTHORIZATION_CODE_MISSING
public static final	AUTHORIZATION_CODE_NOT_ALLOWED
public static final	BAD_XML
public static final	CALLCENTER
public static final	CARDHOLDER_DATA_ALREADY_EXISTS
public static final	COM_PROBLEM
public static final	CONFIGURATION_ERROR
public static final	CREDIT_NOTALLOWED
public static final	CREDIT_PROBLEM
public static final	DATABASE_ERROR
public static final	DECLINED
public static final	DECLINED_CALLCENTER
public static final	ECCO_EXCEPTION
public static final	ECCO_JNI_EXCEPTION
public static final	EXPIRATION_DATE_MISSING
public static final	EXPIRATION_DATE_NOT_ALLOWED
public static final	FILE_NOT_FOUND_EXCEPTION
public static final	INVALID_ABA_ROUTING_NUMBER
public static final	INVALID_ACCOUNT_NUMBER
public static final	INVALID_ADDRESS
public static final	INVALID_ADPAY_PAYMENT_DESIGNATOR

Member Summary	
public static final	INVALID_ADPAY_TRANSACTION_TYPE
public static final	INVALID_AMOUNT
public static final	INVALID_AUTHORIZATION_CODE
public static final	INVALID_CARD_TYPE
public static final	INVALID_CDM_REQUEST
public static final	INVALID_CITY
public static final	INVALID_CLIENT_DATA
public static final	INVALID_EXPIRATION_DATE
public static final	INVALID_MERCHANTID
public static final	INVALID_NETWORK_REPLY
public static final	INVALID_OEP
public static final	INVALID_PASSWORD
public static final	INVALID_PAYMENT_TYPE
public static final	INVALID_SALESTAX
public static final	INVALID_STATE
public static final	INVALID_TELEPHONE_NUMBER
public static final	INVALID_TOKEN
public static final	INVALID_TRACK_DATA
public static final	INVALID_TRANSACTION_TYPE
public static final	INVALID_TRANSACTIONID
public static final	INVALID_USER
public static final	INVALID_ZIPCODE
public static final	IO_EXCEPTION
public static final	MERCHANT_CLOSED
public static final	MISSING_ECCO_PROPERTIES
public static final	MISSING_PAYMENT
public static final	MISSING_SSL_CERTIFY
public static final	NETWORK_DUPLICATE
public static final	NETWORK_PROBLEM_RETRY
public static final	NETWORK_TIMEOUT
public static final	NEVER_DID_LOGON
public static final	NO_RESULTS
public static final	NO_SERVER_CONNECTION
public static final	NO_TRANSACTION_FOUND
public static final	OVER_MAX_TXN_AMOUNT
public static final	PREVIOUSLY_DECLINED
public static final	REFERENCE_NUMBER_MISSING
public static final	SAX_EXCEPTION
public static final	SECURITY_ERROR
public static final	SERVER_DUPLICATE
public static final	STATUS_MISSING
public static final	SUCCESS
public static final	SYS_CONFIG_ERR
public static final	TIMEOUT
public static final	TOKEN_NOT_ALLOWED
public static final	TRANSACTION_NOT_FOUND
public static final	UNKNOWN_REPLY

Member Summary	
public static final	WRONG_QUEUE
public static final	XERCES_MISSING
Constructors	
public	ECCOStatusCodes()

Inherited Member Summary
Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Fields

ABA_ROUTING_NUMBER_MISSING

```
public static final int ABA_ROUTING_NUMBER_MISSING
```

ABA_ROUTING_NUMBER_NOT_ALLOWED

```
public static final int ABA_ROUTING_NUMBER_NOT_ALLOWED
```

ADPAY_PAYMENT_DESIGNATOR_MISSING

```
public static final int ADPAY_PAYMENT_DESIGNATOR_MISSING
```

ADPAY_PAYMENT_DESIGNATOR_NOT_ALLOWED

```
public static final int ADPAY_PAYMENT_DESIGNATOR_NOT_ALLOWED
```

ALREADY_ACKNOWLEDGED

```
public static final int ALREADY_ACKNOWLEDGED
```

ALREADY_LOGGEDON

```
public static final int ALREADY_LOGGED_ON
```

AMOUNT_MISSING

```
public static final int AMOUNT_MISSING
```

AMOUNT_NOT_ALLOWED

```
public static final int AMOUNT_NOT_ALLOWED
```

AUTH_NOT_CAPTURED

```
public static final int AUTH_NOT_CAPTURED
```

AUTHORIZATION_CODE_MISSING

```
public static final int AUTHORIZATION_CODE_MISSING
```

AUTHORIZATION_CODE_NOT_ALLOWED

```
public static final int AUTHORIZATION_CODE_NOT_ALLOWED
```

BAD_XML

```
public static final int BAD_XML
```

CALLCENTER

```
public static final int CALLCENTER
```

CARDHOLDER_DATA_ALREADY_EXISTS

```
public static final int CARDHOLDER_DATA_ALREADY_EXISTS
```

COM_PROBLEM

```
public static final int COM_PROBLEM
```

CONFIGURATION_ERROR

```
public static final int CONFIGURATION_ERROR
```

CREDIT_NOT_ALLOWED

```
public static final int CREDIT_NOT_ALLOWED
```

CREDIT_PROBLEM

```
public static final int CREDIT_PROBLEM
```

DATABASE_ERROR

```
public static final int DATABASE_ERROR
```

DECLINED

```
public static final int DECLINED
```

DECLINED_CALLCENTER

```
public static final int DECLINED_CALL_CENTER
```

ECCO_EXCEPTION

```
public static final int ECCO_EXCEPTION
```

ECCO_JNI_EXCEPTION

```
public static final int ECCO_JNI_EXCEPTION
```

EXPIRATION_DATE_MISSING

```
public static final int EXPIRATION_DATE_MISSING
```

EXPIRATION_DATE_NOT_ALLOWED

```
public static final int EXPIRATION_DATE_NOT_ALLOWED
```

FILE_NOT_FOUND_EXCEPTION

```
public static final int FILE_NOT_FOUND_EXCEPTION
```

INVALID_ABA_ROUTING_NUMBER

```
public static final int INVALID_ABA_ROUTING_NUMBER
```

INVALID_ACCOUNT_NUMBER

```
public static final int INVALID_ACCOUNT_NUMBER
```

INVALID_ADDRESS

```
public static final int INVALID_ADDRESS
```

INVALID_ADPAY_PAYMENT_DESIGNATOR

```
public static final int INVALID_ADPAY_PAYMENT_DESIGNATOR
```

INVALID_ADPAY_TRANSACTION_TYPE

```
public static final int INVALID_ADPAY_TRANSACTION_TYPE
```

INVALID_AMOUNT

```
public static final int INVALID_AMOUNT
```

INVALID_AUTHORIZATION_CODE

```
public static final int INVALID_AUTHORIZATION_CODE
```

INVALID_CARD_TYPE

```
public static final int INVALID_CARD_TYPE
```

INVALID_CDM_REQUEST

```
public static final int INVALID_CDM_REQUEST
```

INVALID_CITY

```
public static final int INVALID_CITY
```

INVALID_CLIENT_DATA

```
public static final int INVALID_CLIENT_DATA
```

INVALID_EXPIRATION_DATE

```
public static final int INVALID_EXPIRATION_DATE
```

INVALID_MERCHANTID

```
public static final int INVALID_MERCHANTID
```

INVALID_NETWORK_REPLY

```
public static final int INVALID_NETWORK_REPLY
```

INVALID_OEP

```
public static final int INVALID_OEP
```

INVALID_PASSWORD

```
public static final int INVALID_PASSWORD
```

INVALID_PAYMENT_TYPE

```
public static final int INVALID_PAYMENT_TYPE
```

INVALID_SALESTAX

```
public static final int INVALID_STATE
```

INVALID_STATE

```
public static final int INVALID_STATE
```

INVALID_TOKEN

```
public static final int INVALID_TOKEN
```

INVALID_TELEPHONE_NUMBER

```
public static final int INVALID_TELEPHONE_NUMBER
```

INVALID_TRACK_DATA

```
public static final int INVALID_TRACK_DATA
```

INVALID_TRANSACTIONID

```
public static final int INVALID_TRANSACTIONID
```

INVALID_TRANSACTION_TYPE

```
public static final int INVALID_TRANSACTION_TYPE
```

INVALID_USER

```
public static final int INVALID_USER
```

INVALID_ZIPCODE

```
public static final int INVALID_ZIPCODE
```

IO_EXCEPTION

```
public static final int IO_EXCEPTION
```

MERCHANT_CLOSED

```
public static final int MERCHANT_CLOSED
```

MISSING_ECCO_PROPERTIES

```
public static final int MISSING_ECCO_PROPERTIES
```

MISSING_PAYMENT

```
public static final int MISSING_PAYMENT
```

MISSING_SSL_CERTIFY

```
public static final int MISSING_SSL_CERTIFY
```

NETWORK_DUPLICATE

```
public static final int NETWORK_DUPLICATE
```

NETWORK_PROBLEM_RETRY

```
public static final int NETWORK_PROBLEM_RETRY
```

NETWORK_TIMEOUT

```
public static final int NETWORK_TIMEOUT
```

NEVER_DID_LOGON

```
public static final int NEVER_DID_LOGON
```

NO_SERVER_CONNECTION

```
public static final int NO_SERVER_CONNECTION
```

NO_TRANSACTION_FOUND

```
public static final int NO_TRANSACTION_FOUND
```

OVER_MAX_TXN_AMOUNT

```
public static final int OVER_MAX_TXN_AMOUNT
```

PREVIOUSLY_DECLINED

```
public static final int PREVIOUSLY_DECLINED
```

SAX_EXCEPTION

```
public static final int SAX_EXCEPTION
```

SECURITY_ERROR

```
public static final int SECURITY_ERROR
```

SERVER_DUPLICATE

```
public static final int SERVER_DUPLICATE
```

STATUS_MISSING

```
public static final int STATUS_MISSING
```

SUCCESS

```
public static final int SUCCESS
```

SYS_CONFIG_ERR

```
public static final int SYS_CONFIG_ERR
```

TIMEOUT

```
public static final int TIMEOUT
```

TOKEN_NOT_ALLOWED

```
public static final int TOKEN_NOT_ALLOWED
```

TRANSACTION_NOT_FOUND

```
public static final int TRANSACTION_NOT_FOUND
```

UNKNOWN_REPLY

```
public static final int UNKNOWN_REPLY
```

XERCES_MISSING

```
public static final int XERCES_MISSING
```

Constructors

ECCOStatusCodes()

```
public ECCOStatusCodes()
```

com.edgil.ecco.eccoapi MonetaryTransactionData

Declaration

```
public class MonetaryTransactionData

java.lang.Object
  |
  +--com.edgil.ecco.eccoapi.MonetaryTransactionData
```

Description

A MonetaryTransactionData object contains the data for a monetary transaction sent to or returned from EdgCapture. The available fields correspond to the transaction data stored in the EdgCapture database. For complete information on transaction data fields in EdgCapture, see "Appendix C: Monetary Transaction Data" on page 189.

Member Summary	
Constructors	
`public`	`MonetaryTransactionData()`
`public`	`MonetaryTransactionData(String, String)`
Methods	
`public void`	`clearAllFields()` Clears all fields in the current MonetaryTransactionData object in order to use it for a new transaction.
`public void`	`clearCardholderData()` Clears all fields related to cardholder data, which is payment account, name, and address information.
`public void`	`clearToken()` Clears the token field which can be used in place of cardholder data in transactions.
`public void`	`copyValueOf(MonetaryTransactionData)` Copies all values from the source MonetaryTransactionData to this MonetaryTransactionData
`public String`	`getABANumber()` Returns the ABA Number for this transaction or null if a ABA Number is not part of the transaction's Account.
`public String`	`getAccountNumber()` Returns the Account Number for this transaction.
`public String`	`getAddressLine1()` Returns address line 1 for this transaction or null if not available.
`public String`	`getAddressLine2()` Returns address line 2 for this transaction or null if not available.
`public String`	`getAddressVerification()` Returns the Address Verification results for a transaction sent out to a credit network for authorization.
`public String`	`getAmount()` Returns the amount of the transaction in EdgCapture.

Member Summary	
`public String`	`getAuthorizationCode()` Returns the authorization code of this transaction.
`public String`	`getAuthorizationDate()` Returns the date of authorization of this transaction.
`public String`	`getAuthorizationString()` Returns the authorization code and date of authorization of this transaction.
`public String`	`getCaptureDate()` Returns the capture date of the transaction as YYYY-MM-DD or null if not available.
`public String`	`getCity()` Returns the city for this transaction or null if not available.
`public String`	`getECI()` Returns the electronic commerce indicator or null if not available.
`public String`	`getExpirationDate()` Returns the expiration date as MMYY or null if not available.
`public String`	`getExpirationMonth()` Returns the expiration month for this transaction or null if an Expiration Date is not part of the transaction's Account.
`public String`	`getExpirationYear()` Returns the expiration year for this transaction or null if an Expiration Date is not part of the transaction's Account.
`public String`	`getFirstName()` Returns the first name or null if not available.
`public int`	`getFraudSecurityResponse()` Returns the message code for the fraud security response from the authorization network.
`public String`	`getLastNameOrCompanyName()` Returns the last or company name or null if not available.
`public String`	`getMerchantId()` Returns the MerchantId associated with this transaction.
`public String`	`getMiddleInitial()` Returns the middle initial of the name or null if not available.
`public String`	`getName()` Returns the name for this transaction or null if not available.
`public String`	`getNotes()` Returns Notes, 80 characters of client specified information.
`public String`	`getOEPId()` Returns the OEPId associated with this transaction.
`public String`	`getPaymentDesignator()` Returns the payment designator for this transaction or null if a payment designator is not part of the transaction's Account.
`public String`	`getPaymentType()` Returns the payment type for this transaction.
`public String`	`getProcessingState()` Returns the processing state of the transaction or null if not available.
`public String`	`getQueueDateTime()` Returns the datetime of entry for the transaction's queue in EdgCapture.

Member Summary	
`public String`	`getQueueName()` Returns the name of the transaction’s queue in EdgCapture.
`public String`	`getQueueString()` Returns the name and date of entry for the transaction’s queue in EdgCapture.
`public String`	`getReferenceNumber()` Returns the Reference Number generated by the Credit Network or EdgCapture at authorization time.
`public String`	`getSalesTax()` Returns the sales tax amount.
`public String`	`getSource()` Returns the Process source or null if not available.
`public String`	`getStateOrProvince()` Returns the customer address state or province for this transaction or null if not available.
`public String`	`getTelephoneNumber()` Returns the telephone number for this transaction or null if not available.
`public Strin`	`getToken()` Returns the token for this transaction or null.
`public String`	`getTransactionId()` Returns the Transaction Id.
`public int`	`getTransactionStatusCode()` Returns message code for the status of the transaction in EdgCapture.
`public String`	`getTransactionStatusMessage()` Returns message text for the status of the transaction in EdgCapture.
`public String`	`getTransactionStatusString()` Returns message code and text for the status of the transaction in EdgCapture.
`public String`	`getTransactionType()` Returns the transaction type.
`public String`	`getUserData1()` Returns UserData1, 40 characters of client specified information.
`public String`	`getUserData10()` Returns UserData10, 40 characters of client specified information.
`public String`	`getUserData2()` Returns UserData2, 40 characters of client specified information.
`public String`	`getUserData3()` Returns UserData3, 40 characters of client specified information.
`public String`	`getUserData4()` Returns UserData4, 40 characters of client specified information.
`public String`	`getUserData5()` Returns UserData5, 40 characters of client specified information.
`public String`	`getUserData6()` Returns UserData6, 40 characters of client specified information.
`public String`	`getUserData7()` Returns UserData7, 40 characters of client specified information.
`public String`	`getUserData8()` Returns UserData8, 40 characters of client specified information.

Member Summary	
`public String`	`getUserData9()` Returns UserData9, 40 characters of client specified information.
`public String`	`getZipCode()` Returns zipcode for this transaction or null if not available.
`public void`	`setABANumber(String)` Sets the ABA Number as part of the AUTHORIZE REQUIRED payment information.
`public void`	`setAccountNumber(String)` Sets the Account Number as part of the AUTHORIZE and CREDIT REQUIRED payment information.
`public void`	`setAddress(String, String, String)` Sets the OPTIONAL address values for city, state or province and zipcode.
`public void`	`setAddress(String, String, String, String)` Sets the OPTIONAL address values for address line one, city, state or province and zipcode.
`public void`	`setAddressLine1(String)` Sets the value of OPTIONAL address line one.
`public void`	`setAddressLine2(String)` Sets the value of OPTIONAL address line two.
`public void`	`setAmount(String)` Sets the value for the AUTHORIZE and CREDIT REQUIRED amount of the transaction.
`public void`	`setCheckDraft(String, String, String)` Sets the AUTHORIZE and CREDIT REQUIRED payment information as a check draft.
`public void`	`setCity(String)` Sets the value of OPTIONAL city.
`public void`	`setCreditCard(String, String, String)` Sets the AUTHORIZE and CREDIT REQUIRED payment information as a credit card.
`public void`	`setCreditCard(String, String, String, String)` Sets the AUTHORIZE and CREDIT REQUIRED payment information as a credit card.
`public void`	`setDirectDebit(String, String, String)` Sets the AUTHORIZE and CREDIT REQUIRED payment information as direct debit.
`public void`	`setECI(String)` Sets the value for the REQUIRED electronic commerce indicator.
`public void`	`setExpirationDate(String, String)` Sets the Expiration Date as part of the AUTHORIZE and CREDIT REQUIRED payment information.
`public void`	`setExpirationMonth(String)` Sets the Expiration Month as part of the AUTHORIZE and CREDIT REQUIRED payment information.
`public void`	`setExpirationYear(String)` Sets the Expiration Year as part of the AUTHORIZE and CREDIT REQUIRED payment information.

Member Summary	
`public void`	`setFraudSecurityValue(String)` Sets the value for the OPTIONAL fraud security value.
`public void`	`setMerchantId(String)` Sets the value for the REQUIRED MerchantId.
`public void`	`setName(String)` Sets the value of the OPTIONAL customer name: companyName or lastName Maximum size is 60.
`public void`	`setName(String, String)` Sets the value of the OPTIONAL customer name: firstName, lastName
`public void`	`setName(String, String, String)` Sets the value of the OPTIONAL customer name: firstName, middleInitial, lastName.
`public void`	`setNotes(String)` Sets the value of OPTIONAL notes.
`public void`	`setOEPId(String)` Sets the value for the REQUIRED OEPId.
`public void`	`setPaymentDesignator(String)` Sets the Payment Designator as part of the AUTHORIZE REQUIRED payment information.
`public void`	`setPaymentType(String)` Sets the Payment Type as part of the AUTHORIZE and CREDIT REQUIRED payment information.
`public void`	`setPriority(int)` Resets the priority for logging by the MonetaryTransactionData class
`public void`	`setReferenceNumber(String)` Sets the value of reference number as part of VOID REQUIRED payment information.
`public void`	`setSalesTax(String)` Sets the value of OPTIONAL sales tax.
`public void`	`setStateOrProvince(String)` Sets the value of OPTIONAL state or province.
`public void`	`setTelephone(String)` Sets the values of OPTIONAL telephone number.
`public void`	`setToken(String)` Sets the value of the token.
`public void`	`setTransactionId(String)` Sets the value of the REQUIRED TransactionId which must be unique within the Merchant and OEP.
`public void`	`setUserData1(String)` Sets the value of OPTIONAL UserData1.
`public void`	`setUserData10(String)` Sets the value of OPTIONAL UserData10.
`public void`	`setUserData2(String)` Sets the value of OPTIONAL UserData2.
`public void`	`setUserData3(String)` Sets the value of OPTIONAL UserData3.
`public void`	`setUserData4(String)` Sets the value of OPTIONAL UserData4.

Member Summary	
`public void`	`setUserData5(String)` Sets the value of OPTIONAL UserData5.
`public void`	`setUserData6(String)` Sets the value of OPTIONAL UserData6.
`public void`	`setUserData7(String)` Sets the value of OPTIONAL UserData7.
`public void`	`setUserData8(String)` Sets the value of OPTIONAL UserData8.
`public void`	`setUserData9(String)` Sets the value of OPTIONAL UserData9.
`public void`	`setZipCode(String)` Sets the value of OPTIONAL zipcode.

Inherited Member Summary
Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructors

MonetaryTransactionData()

```
public MonetaryTransactionData()
```

Constructs a MonetaryTransactionData object that does not log.

MonetaryTransactionData(String, String)

```
public MonetaryTransactionData(String sessionLoggingID
                String eccoPropertiesFile)
```

Constructs a MonetaryTransactionData object that logs using the specified ID and properties file.

Methods

clearAllFields()

```
public void clearAllFields()
```

Clears all fields in the current MonetaryTransactionData object in order to use it for a new transaction.

clearCardholderData)

```
public void clearCardholderData()
```

Clears the cardholder data fields, which include account, name and address information. The token field is unchanged.

clearToken()

```
public void clearToken()
```

Clears the token field, leaves the cardholder data fields unchanged.

copyValueOf(MonetaryTransactionData)

```
public void copyValueOf(MonetaryTransactionData
              copySourceMonetaryTransactionData)
```

Copies all values from the source MonetaryTransactionData to this MonetaryTransactionData

Parameters:

`copySourceMonetaryTransactionData` - Source for copy

getABANumber()

```
public String getABANumber()
```

Returns the ABA Number for this transaction or null if a ABA Number is not part of the transaction’s Account. Maximum size is 9.

getAccountNumber()

```
public String getAccountNumber()
```

Returns the Account Number for this transaction. Maximum size is 48.

getAddressLine1()

```
public String getAddressLine1()
```

Returns address line 1 for this transaction or null if not available. Maximum size is 100.

getAddressLine2()

```
public String getAddressLine2()
```

Returns address line 2 for this transaction or null if not available. Maximum size is 100.

getAddressVerification()

```
public String getAddressVerification()
```

Returns the address verification results for a transaction sent out to a credit network for authorization. A blank will be returned when address verification results are not available. Size is 1.

getAmount()

```
public String getAmount()
```

Returns the amount of the transaction in EdgCapture. The amount is always positive whether the transaction is a Sale or a Credit. Maximum value is 12.2

getAuthorizationCode()

```
public String getAuthorizationCode()
```

Returns the authorization code of this transaction. Maximum size is 10.

getAuthorizationDate()

```
public String getAuthorizationDate()
```

Returns the date this transaction was authorized by the credit network in YYYY-MM-DD format.

getAuthorizationString()

```
public String getAuthorizationString()
```

Returns the authorization code and date of authorization of this transaction. The date of authorization is year-month-day in the form YYYY-MM-DD. Maximum size for the authorization code is 10.

getCaptureDate()

```
public String getCaptureDate()
```

Returns the capture date of the transaction as YYYY-MM-DD or null if not available.

getCity()

```
public String getCity()
```

Returns the city for this transaction or null if not available. Maximum size is 50.

getECI()

```
public String getECI()
```

Returns the one-character electronic commerce indicator.

getExpirationDate()

```
public String getExpirationDate()
```

Returns the expiration date as MMYY or null if the not available.

getExpirationMonth()

```
public String getExpirationMonth()
```

Returns the 2-digit expiration month for this transaction or null if an Expiration Date is not part of the transaction's Account. Values are 01-12.

getExpirationYear()

```
public String getExpirationYear()
```

Returns the 2-digit expiration year for this transaction or null if an Expiration Date is not part of the transaction's Account.

getFirstName()

```
public String getFirstName()
```

Returns the first name or null if not available. Maximum size is 60.

getFraudSecurityResponse()

```
public String getFraudSecurityResponse()
```

Returns the fraud security response code from the authorization network.

getLastNameOrCompanyName()

```
public String getLastNameOrCompanyName()
```

Returns the last or company name or null if not available. Maximum size is 60.

getMerchantId()

```
public String getMerchantId()
```

Returns the MerchantId associated with this transaction. The MerchantId is a fixed String identifying the Merchant to EdgCapture. The Merchant represents the bank account for the transaction funds. Maximum size is 2.

getMiddleInitial()

```
public String getMiddleInitial()
```

Returns the middle initial of the name or null if not available. Maximum size is 2.

getName()

```
public String getName()
```

Returns the name for this transaction or null if not available. Name is FirstName MiddleInitial LastName. Maximum size is 186.

getNotes()

```
public String getNotes()
```

Returns Notes, 80 characters of client specified information.

getOEPId()

```
public String getOEPId()
```

Returns the OEPId associated with this transaction. The OEPId is a fixed string identifying a subaccount for a Merchant. Maximum size is 2.

getPaymentDesignator()

`public String **getPaymentDesignator**()`

Returns the payment designator for this transaction or null if a payment designator is not part of the transaction's Account. Values are ECCOClient.CHECK_DESIGNATOR_CORPORATE or ECCOClient.CHECK_DESIGNATOR_CONSUMER.

getPaymentType()

`public String **getPaymentType**()`

Returns the payment type for this transaction. Maximum size is 30. ECCOClient - PAY_AMERICAN_EXPRESS,PAY_CARTE_BLANCHE, PAY_CASH, PAY_CHECK_DRAFT, PAY_DINERS_CLUB, PAY_DIRECT_DEBIT, PAY_DISCOVER, PAY_MASTER_CARD, PAY_PAPER_CHECK, PAY_VISA

getQueueDateTime()

`public String **getQueueDateTime**()`

Returns the datetime of entry for the transaction's queue in EdgCapture. The date of entry is year-month-day hour-minute-second in the form YYYY-MM-DD HH-MM-SS. Transactions move from queue to queue as they are processed on EdgCapture.

getQueueName()

`public String **getQueueName**()`

Returns the name of the transaction's queue in EdgCapture. Transactions move from queue to queue as they are processed on EdgCapture. ECCOClient Set of queues: QUEUE_AUTHORIZED, QUEUE_CAPTURED, QUEUE_DECLINED, QUEUE_ERROR, QUEUE_HELD, QUEUE_INPUT, QUEUE_MARKED_FOR_CAPTURE, QUEUE_RECORDED_ONLY, QUEUE_UNKNOWN_DISPOSITION, QUEUE_VOIDED Maximum size is 30.

getQueueString()

`public String **getQueueString**()`

Returns the name and datetime of entry for the transaction's queue in EdgCapture. The datetime of entry is year-month-day hour-minute-second in the form YYYY-MM-DD HH-MM-SS. Transactions move from queue to queue as they are processed on EdgCapture. ECCOClient Set of queues: QUEUE_AUTHORIZED, QUEUE_CAPTURED, QUEUE_DECLINED, QUEUE_ERROR, QUEUE_HELD, QUEUE_INPUT, QUEUE_MARKED_FOR_CAPTURE, QUEUE_RECORDED_ONLY, QUEUE_UNKNOWN_DISPOSITION, QUEUE_VOIDED Maximum size is 30.

getReferenceNumber()

`public String **getReferenceNumber**()`

Returns the Reference Number generated by the Credit Network or EdgCapture at authorization time. Maximum size is 10.

getSalesTax()

`public String` **`getSalesTax`**`()`

Returns the optional sales tax of the transaction in nnnn.nn format. Maximum size is 12,2. Note that the Amount contains the total transaction amount, including sales tax.

getSource()

`public String` **`getSource`**`()`

Returns the Process source or null if not available. Maximum size is 40.

getStateOrProvince()

`public String` **`getStateOrProvince`**`()`

Returns the customer address state or province for this transaction or null if not available. Maximum size is 2.

getTelephoneNumber()

`public String` **`getTelephoneNumber`**`()`

Returns the telephone number for this transaction or null if not available. Maximum size is 16.

getToken()

`public String` **`getToken`**`()`

Returns the nine character token for this trancaction, or null if there is no token.

getTransactionId()

`public String` **`getTransactionId`**`()`

Returns the Transaction Id. Maximum size is 30.

getTransactionStatusCode()

`public int` **`getTransactionStatusCode`**`()`

Returns message code for the status of the transaction in EdgCapture. Return -1 if code not available.

getTransactionStatusMessage()

`public String` **`getTransactionStatusMessage`**`()`

Returns message text for the status of the transaction in EdgCapture. Maximum size is determined by the longest message in ECCOStatusMessages.properties.

getTransactionStatusString()

```
public String getTransactionStatusString()
```

Returns message code and text for the status of the transaction in EdgCapture. The format is the message code, a blank, the message string. Maximum size is determined by the longest message in ECCOStatusMessages.properties plus 5.

getTransactionType()

```
public String getTransactionType()
```

Returns the transaction type from ECCOClient: TRAN_AUTHORIZE, TRAN_CAPTURE_HELD_TRANSACTION, TRAN_CREDIT, TRAN_DECLINE_HELD_TRANSACTION, TRAN_MARKED_FOR_CAPTURE, TRAN_RECORD_ONLY, TRAN_VOID. Maximum size is 30.

getUserData1()

```
public String getUserData1()
```

Returns UserData1, 40 characters of client specified information.

getUserData10()

```
public String getUserData10()
```

Returns UserData10, 40 characters of client specified information.

getUserData2()

```
public String getUserData2()
```

Returns UserData2, 40 characters of client specified information.

getUserData3()

```
public String getUserData3()
```

Returns UserData3, 40 characters of client specified information.

getUserData4()

```
public String getUserData4()
```

Returns UserData4, 40 characters of client specified information.

getUserData5()

```
public String getUserData5()
```

Returns UserData5, 40 characters of client specified information.

getUserData6()

```
public String getUserData6()
```

Returns UserData6, 40 characters of client specified information.

getUserData7()

`public String getUserData7()`

Returns UserData7, 40 characters of client specified information.

getUserData8()

`public String getUserData8()`

Returns UserData8, 40 characters of client specified information.

getUserData9()

`public String getUserData9()`

Returns UserData9, 40 characters of client specified information.

getZipCode()

`public String getZipCode()`

Returns zipcode for this transaction or null if not available.

setABANumber(String)

`public void setABANumber(String aBANumber)`

Sets the ABA Number as part of the REQUIRED payment information for an authorize request. When ABA Number is set individually, the other related fields must also be set: AccountNumber, PaymentDesignator, PaymentType. ABA Number is used with Payment Types PAY_DIRECT_DEBIT and PAY_CHECK_DRAFT. Maximum size is 9.

Parameters:

`aBANumber` - The bank routing number for a checking account.

setAccountNumber(String)

`public void setAccountNumber(String accountNumber)`

Sets the Account Number as part of the REQUIRED payment information for an authorize, credit, or record only request. When payment type is set individually, the other related fields must also be set. When Payment type is PAY_CREDIT_CARD or a specific card type (PAY_AMERICAN_EXPRESS, PAY_CARTE_BLANCHE, PAY_DINERS_CLUB, PAY_DISCOVER, PAY_MASTER_CARD, PAY_VISA), Payment Type and ExpirationDate must also be set for authorize or credit requests; record only requests do not require ExpirationDate. When Payment Type is PAY_DIRECT_DEBIT or PAY_CHECK_DRAFT, Payment Type, ABANumber and PaymentDesignator must also be set. Maximum size is 48.

Parameters:

`accountNumber` - For credit cards, the credit card number, for direct debit and check draft, the account number.

setAddress(String, String, String)

```
public void setAddress(String city,
               String stateOrProvince,
               String zipCode)
```

Sets the OPTIONAL address values for city, state or province and zipcode.

Parameters:

`city` - The customer city. Maximum size is 50.

`stateOrProvince` - The state or province. Maximum size is 2.

`zipCode` - The zipCode or postal code. Maximum size is 12. Only alphas and numerics permitted.

setAddress(String, String, String, String)

```
public void setAddress(String addressLine1,
               String city,
               String stateOrProvince,
               String zipCode)
```

Sets the OPTIONAL address values for address line one, city, state or province and zipcode.

Parameters:

`addressLine1` - The first line of the customer's address. Maximum size is 100.

`city` - The customer city. Maximum size is 50.

`stateOrProvince` - The state or province. Maximum size is 2.

`zipCode` - The zipCode or postal code. Maximum size is 12. Only alphas and numerics permitted.

set_AddressLine1(String)

```
public void setAddressLine1(String addressLine1)
```

Sets the value of OPTIONAL address line one. Maximum size is 100.

Parameters:

`addressLine1` - The first line of the customer's address. Maximum size is 100.

set_AddressLine2(String)

```
public void setAddressLine2(String addressLine2)
```

Sets the value of OPTIONAL address line two. Maximum size is 100.

Parameters:

`addressLine2` - The second line of the customer's address. Maximum size is 100.

setAmount

```
public void setAmount(String amount)
```

Sets the REQUIRED amount of the transaction for an authorize, credit, or record only request. The amount must be entered as a decimal value in the format nnn.nn. The decimal point and the two digits after the decimal point are required. The amount is always positive whether the transaction is a Sale or a Credit. Maximum size is 12,2.

Parameters:

`amount`- Amount of the transaction in the format: nnnnn.nn.

setCheckDraft(String, String, String)

```
public void setCheckDraft(String accountNumber,
              String aBANumber,
              String paymentDesignator)
```

Sets all REQUIRED check draft payment information for an authorize or credit request.

Parameters:

`accountNumber` - The check number for the check draft. Maximum size is 48.

`aBANumber` - The bank routing number for the check draft. Maximum size is 9.

`paymentDesignator` - The checking account type which must be CHECK_DESIGNATOR_CORPORATE or CHECK_DESIGNATOR_CONSUMER

setCity(String)

```
public void setCity(String city)
```

Sets the value of OPTIONAL city. Maximum size is 50.

Parameters:

`city` - The customer city. Maximum size is 50.

setCreditCard(String, String, String)

```
public void setCreditCard(String accountNumber,
              String expirationMonth,
              String expirationYear)
```

Sets all REQUIRED credit card payment information for an authorize or credit request.

Parameters:

`accountNumber` - The account number for the credit card. Maximum size is 48.

`expirationMonth` - The expiration month as two digits: nn (01 to 12)

`expirationYear` - The expiration year as two digits: nn

setCreditCard(String, String, String, String)

```
public void setCreditCard(String accountNumber,
              String expirationMonth,
              String expirationYear,
              String paymentType)
```

Sets all REQUIRED credit card payment information for an authorize or credit request. Also sets OPTIONAL payment type.

Parameters:

`accountNumber` - The account number for the credit card. Maximum size is 48.

`expirationMonth` - The expiration month as two digits: nn (01 to 12)

`expirationYear` - The expiration year as two digits: nn

`paymentType` - User selected payment type for cross validation with account number. ECCOClient payment types: PAY_AMERICAN_EXPRESS, PAY_CARTE_BLANCHE PAY_DINERS_CLUB, PAY_DISCOVER, PAY_MASTER_CARD, PAY_VISA

setDirectDebit(String, String, String)

```
public void setDirectDebit(String accountNumber,
              String aBANumber,
              String paymentDesignator)
```

Sets all REQUIRED direct debit payment information for an authorize or credit request.

Parameters:

`accountNumber` - The account number for direct debit. Maximum size is 48.

`aBANumber` - The bank routing number for direct debit. Maximum size is 9.

`paymentDesignator` - The checking account type which must be CHECK_DESIGNATOR_CORPORATE or CHECK_DESIGNATOR_CONSUMER

setECI(String)

```
public void setECI(String ECI)
```

Sets the electronic commerce indicator REQUIRED for authorize requests. It becomes part of the message sent by EdgCapture to the processor network.

Parameters:

`ECI` - The one-character ECI, which must be one of the following values. Note that 6 is the most likely value for web-based ECCO applications. 4 and 5 are not yet implemented.

0 - Unknown
1 - Mail Order/Telephone Order (single, non-recurring transaction)
2 - Mail Order/Telephone Order (recurring periodic transaction, such as subscription)
3 - Mail Order/Telephone Order (installment)
4 - Secure Electronic Transaction (SET) protocol using cardholder certificate management
5 - SET with no cardholder certificate management, but includes merchant certificate
6 - Non-SET, channel-encrypted security such as SSL (Web site using SSL)
7 - Non-SET, non-channel-encrypted security (clear text e-mail)

setExpirationDate(String, String)

```
public void setExpirationDate(String expirationMonth,
              String expirationYear)
```

Sets the Expiration Date as part of the AUTHORIZE and CREDIT REQUIRED payment information. When the Expiration Date is set individually, the other related fields must also be set: AccountNumber, PaymentType. Expiration Date is used with Payment Type PAY_CREDIT_CARD and the specific card types: PAY_AMERICAN_EXPRESS, PAY_CARTE_BLANCHE, PAY_DINERS_CLUB, PAY_DISCOVER, PAY_MASTER_CARD, PAY_VISA.

Parameters:

`expirationMonth` - The expiration month as two digits: nn (01 to 12)

`expirationYear` - The expiration year as two digits: nn

setExpirationMonth(String)

`public void setExpirationMonth(String expirationMonth)`

Sets the Expiration Month as part of the REQUIRED payment information for an authorize or credit request. When the Expiration Month is set individually, the other related fields must also be set: Expiration Year, AccountNumber, PaymentType. Expiration Month is used with Payment Type PAY_CREDIT_CARD and the specific card types: PAY_AMERICAN_EXPRESS, PAY_CARTE_BLANCHE, PAY_DINERS_CLUB, PAY_DISCOVER, PAY_MASTER_CARD, PAY_VISA.

Parameters:

`expirationMonth` - The expiration month as two digits: nn (1 to 12)

setExpirationYear(String)

`public void setExpirationYear(String expirationYear)`

Sets the Expiration Year as part of the REQUIRED payment information for an authorize or credit request. When the Expiration Year is set individually, the other related fields must also be set: Expiration Month, AccountNumber, PaymentType. Expiration Year is used with Payment Type PAY_CREDIT_CARD and the specific card types: PAY_AMERICAN_EXPRESS, PAY_CARTE_BLANCHE, PAY_DINERS_CLUB, PAY_DISCOVER, PAY_MASTER_CARD, PAY_VISA.

Parameters:

`expirationYear` - The expiration year as two digits: nn

setFraudSecurityValue(String)

`public void setFraudSecurityValue(String fraudSecurityValue)`

Sets the value for the OPTIONAL fraud security value, a 3- or 4-digit code separate from the account number found on Visa, Master Card, American Express, and Discover cards. The value provides additional security during authorization for merchants in card-not-present transactions. Note that in accordance with card issuer's regulations, this value is not stored in the database for later retrieval.

Parameters:

`fraudSecurityValue` - The 3- or 4-digit code from the credit card.

setMerchantId(String)

`public void setMerchantId(String merchantId)`

Sets the value for the REQUIRED MerchantId. The MerchantId is a fixed string identifying the destination bank account to EdgCapture. Maximum size is 2.

Parameters:

`merchantId` - A fixed string identifying the destination bank account to EdgCapture.

setName(String)

```
public void setName(String lastNameOrCompany)
```

Sets the value of the OPTIONAL customer name to either the company name or last name. Maximum size is 60.

Parameters:

`lastNameorCompany` - Last name of customer or company name.

setName(String, String)

```
public void setName(String firstName,
                    String lastName)
```

Sets the value of the OPTIONAL customer name: firstName, lastName

Parameters:

`firstName` - The first name of the customer. Maximum size is 60.

`lastName` - The last name of the customer. Maximum size is 60.

setName(String, String, String)

```
public void setName(String firstName,
                    String middleInitial,
                    String lastName)
```

Sets the value of the OPTIONAL customer name: firstName, middleInitial, lastName.

Parameters:

`firstName` - The first name of the customer. Maximum size is 60.

`middleInitial` - The middle initial of the customer. Maximum size is 2.

`lastName` - The last name of the customer. Maximum size is 60.

setNotes(String)

```
public void setNotes(String notes)
```

Sets the value of OPTIONAL notes. Notes is 80 characters of client specified information kept on EdgCapture where it may be viewed, reported on or returned to ECCO with other transaction information.

Parameters:

`notes` - Up to 80 characters of transaction notes.

setOEPId(String)

```
public void setOEPId(String oepId)
```

Sets the value for the REQUIRED OEPId. The OEPId is a fixed string identifying the client subaccount to EdgCapture. Maximum size is 2.

Parameters:

`oepId` - Fixed string identifying the client subaccount. Maximum size is 2.

setPaymentDesignator(String)

`public void **setPaymentDesignator**(String paymentDesignator)`

Sets the Payment Designator as part of the REQUIRED payment information for an authorize or credit request for a direct debit or checking account. When Payment Designator is set individually, the other related fields must also be set: AccountNumber, ABANumber, PaymentType. Payment Designator is used with Payment Types PAY_DIRECT_DEBIT and PAY_CHECK_DRAFT.

Parameters:

`paymentDesignator` - The checking account type which must be CHECK_DESIGNATOR_CORPORATE or CHECK_DESIGNATOR_CONSUMER

setPaymentType(String)

`public void **setPaymentType**(String paymentType)`

Sets the Payment Type as part of the REQUIRED payment information for an authorize or credit request. When payment type is set individually, the other related fields must also be set. When Payment type is PAY_CREDIT_CARD or a specific card type (PAY_AMERICAN_EXPRESS, PAY_CARTE_BLANCHE, PAY_DINERS_CLUB, PAY_DISCOVER, PAY_MASTER_CARD, PAY_VISA), AccountNumber and ExpirationDate must also be set. When Payment Type is PAY_DIRECT_DEBIT or PAY_CHECK_DRAFT, AccountNumber, ABANumber and PaymentDesignator must also be set.

Parameters:

`paymentType` - Payment type which must be one of the following:

PAY_CREDIT_CARD,PAY_AMERICAN_EXPRESS, PAY_CARTE_BLANCHE, PAY_DINERS_CLUB, PAY_DISCOVER, PAY_MASTER_CARD, PAY_VISA, PAY_DIRECT_DEBIT, or PAY_CHECK_DRAFT.

setPriority(int)

`public void **setPriority**(int priority)`

Resets the priority for logging by the MonetaryTransactionData class.

Parameters:

`priority` - Logging priority level: INFO, WARN, or ERROR

setReferenceNumber(String)

`public void **setReferenceNumber**(String referenceNumber)`

Sets the value of the Reference Number REQUIRED for a VOID request.

Parameters:

`referenceNumber` - Reference number returned from EdgCapture after processing.

setSalesTax(String)

`public void **setSalesTax**(String salesTax)`

Sets the value of the OPTIONAL sales tax amount. Values are used for settlement on commercial card transactions. Note that Amount contains the total transaction amount, including sales tax.

Parameters:

`salesTax` - Amount of the sales tax entered as a decimal value: nnnn.nn.

The decimal point and the two digits after the decimal point are required. Note that 0 may be expressed as .00 or 0.00. Maximum size is 12,2.

setStateOrProvince(String)

`public void setStateOrProvince(String stateOrProvince)`

Sets the value of OPTIONAL state or province code. Maximum size is 2.

Parameters:

`stateOrProvince` - Two-character state or province code.

setTelephone(String)

`public void setTelephone(String phoneNumber)`

Sets the values of OPTIONAL telephone number. Maximum size is 16. No formatting is required.

Parameters:

`phoneNumber` - Unformatted phone number string. Maximum size is 16.

setToken(String)

`public void setToken(String token)`

Sets the value of the token field. Tokens are nine character strings representing cardholder data.

Parameters:

`token` - nine character token or empty string.

setTransactionId(String)

`public void setTransactionId(String transactionId)`

Sets the value of the REQUIRED TransactionId which must be unique within the Merchant and OEP. Maximum size is 30.

Parameters:

`transactionId` - String identifying transaction. Maximum size is 30.

setUserData1(String)

`public void setUserData1(String userData)`

Sets the value of OPTIONAL UserData1. UserData1 is 40 characters of client specified information kept on EdgCapture where it may be viewed, reported on or returned to ECCO with other transaction information.

Parameters:

`userData` - Up to 40 characters of user-specified data.

setUserData10(String)

`public void setUserData10(String userData)`

Sets the value of OPTIONAL UserData10. UserData10 is 40 characters of client specified information kept on EdgCapture where it may be viewed, reported on or returned to ECCO with other transaction information.

Parameters:
`userData` - Up to 40 characters of user-specified data.

setUserData2(String)

```
public void setUserData2(String userData)
```

Sets the value of OPTIONAL UserData2. UserData2 is 40 characters of client specified information kept on EdgCapture where it may be viewed, reported on or returned to ECCO with other transaction information.

Parameters:
`userData` - Up to 40 characters of user-specified data.

setUserData3(String)

```
public void setUserData3(String userData)
```

Sets the value of OPTIONAL UserData3. UserData3 is 40 characters of client specified information kept on EdgCapture where it may be viewed, reported on or returned to ECCO with other transaction information.

Parameters:
`userData` - Up to 40 characters of user-specified data.

setUserData4(String)

```
public void setUserData4(String userData)
```

Sets the value of OPTIONAL UserData4. UserData3 is 40 characters of client specified information kept on EdgCapture where it may be viewed, reported on or returned to ECCO with other transaction information.

Parameters:
`userData` - Up to 40 characters of user-specified data.

setUserData5(String)

```
public void setUserData5(String userData)
```

Sets the value of OPTIONAL UserData5. UserData5 is 40 characters of client specified information kept on EdgCapture where it may be viewed, reported on or returned to ECCO with other transaction information.

Parameters:
`userData` - Up to 40 characters of user-specified data.

setUserData6(String)

```
public void setUserData6(String userData)
```

Sets the value of OPTIONAL UserData6. UserData6 is 40 characters of client specified information kept on EdgCapture where it may be viewed, reported on or returned to ECCO with other transaction information.

Parameters:
`userData` - Up to 40 characters of user-specified data.

setUserData7(String)

```
public void setUserData7(String userData)
```

Sets the value of OPTIONAL UserData7. UserData7 is 40 characters of client specified information kept on EdgCapture where it may be viewed, reported on or returned to ECCO with other transaction information.

Parameters:
`userData` - Up to 40 characters of user-specified data.

setUserData8(String)

```
public void setUserData8(String userData)
```

Sets the value of OPTIONAL UserData8. UserData8 is 40 characters of client specified information kept on EdgCapture where it may be viewed, reported on or returned to ECCO with other transaction information.

Parameters:
`userData` - Up to 40 characters of user-specified data.

setUserData9(String)

```
public void setUserData9(String userData)
```

Sets the value of OPTIONAL UserData9. UserData9 is 40 characters of client specified information kept on EdgCapture where it may be viewed, reported on or returned to ECCO with other transaction information.

Parameters:
`userData` - Up to 40 characters of user-specified data.

setZipCode(String)

```
public void setZipCode(String zipCode)
```

Sets the value of OPTIONAL zipcode. Note, however, that zip code is required for address verification. Only alphanumeric characters are permitted. Maximum size is 12.

Parameters:
`zipCode` - Zip code or postal code, expressed as alphanumeric string. Maximum size is 12.

com.edgil.ecco.eccoapi CardHolderData

Declaration

```
public class CardHolderData

java.lang.Object
  |
  +--com.edgil.ecco.eccoapi.CardHolderdata
```

Description

CardholderData represents cardholder data for use in CDM ECCOClient requests. Cardholder data includes all account information, name, and address data. This is separate from transaction data.

Member Summary	
Constructors	
public	`CardHolderData()`
public	`CardHolderData(String, String)`
Methods	
public void	`clearAllFields()` Clears all fields previously set data fields
public void	`clearCardholderData()` Clears the cardholder data fields but leaves the token field unchanged.
public void	`clearToken()` Clears the token field but leaves the cardholder data unchanged
public void	`copyValueOf(CardHolderdata)` Copies all values from the source MonetaryTransactionData to this MonetaryTransactionData
public String	`getABANumber()` Returns the ABA Number for this transaction or null if a ABA Number is not part of the transaction's Account.
public String	`getAccountNumber()` Returns the Account Number for this transaction.
public String	`getAddressLine1()` Returns address line 1 for this transaction or null if not available.
public String	`getAddressLine2()` Returns address line 2 for this transaction or null if not available.
public String	`getCity()` Returns the city for this transaction or null if not available.
public String	`getExpirationDate()` Returns the expiration date as MMYY or null if not available.
public String	`getExpirationMonth()` Returns the expiration month for this transaction or null if an Expiration Date is not part of the transaction's Account.
public String	`getExpirationYear()` Returns the expiration year for this transaction or null if an Expiration Date is not part of the transaction's Account.

Member Summary	
`public String`	`getFirstName()` Returns the first name or null if not available.
`public String`	`getLastNameOrCompanyName()` Returns the last or company name or null if not available.
`public String`	`getMiddleInitial()` Returns the middle initial of the name or null if not available.
`public String`	`getName()` Returns the name for this transaction or null if not available.
`public String`	`getPaymentDesignator()` Returns the payment designator for this transaction or null if a payment designator is not part of the transaction's Account.
`public String`	`getPaymentType()` Returns the payment type for this transaction.
`public String`	`getStateOrProvince()` Returns the customer address state or province for this transaction or null if not available.
`public String`	`getTelephoneNumber()` Returns the telephone number for this transaction or null if not available.
`public String`	`getToken()` Returns the token for this transaction or null.
`public String`	`getZipCode()` Returns zipcode for this transaction or null if not available.
`public void`	`setABANumber(String)` Sets the ABA Number as part of the AUTHORIZE REQUIRED payment information.
`public void`	`setAccountNumber(String)` Sets the Account Number as part of the AUTHORIZE and CREDIT REQUIRED payment information.
`public void`	`setAddress(String, String, String)` Sets the OPTIONAL address values for city, state or province and zipcode.
`public void`	`setAddress(String, String, String, String)` Sets the OPTIONAL address values for address line one, city, state or province and zipcode.
`public void`	`setAddressLine1(String)` Sets the value of OPTIONAL address line one.
`public void`	`setAddressLine2(String)` Sets the value of OPTIONAL address line two.
`public void`	`setCheckDraft(String, String, String)` Sets the AUTHORIZE and CREDIT REQUIRED payment information as a check draft.
`public void`	`setCity(String)` Sets the value of OPTIONAL city.
`public void`	`setCreditCard(String, String, String)` Sets the AUTHORIZE and CREDIT REQUIRED payment information as a credit card.

Member Summary	
public void	`setCreditCard(String, String, String, String)` Sets the AUTHORIZE and CREDIT REQUIRED payment information as a credit card.
public void	`setDirectDebit(String, String, String)` Sets the AUTHORIZE and CREDIT REQUIRED payment information as direct debit.
public void	`setExpirationDate(String, String)` Sets the Expiration Date as part of the AUTHORIZE and CREDIT REQUIRED payment information.
public void	`setExpirationMonth(String)` Sets the Expiration Month as part of the AUTHORIZE and CREDIT REQUIRED payment information.
public void	`setExpirationYear(String)` Sets the Expiration Year as part of the AUTHORIZE and CREDIT REQUIRED payment information.
public void	`setName(String)` Sets the value of the OPTIONAL customer name: companyName or lastName Maximum size is 60.
public void	`setName(String, String)` Sets the value of the OPTIONAL customer name: firstName, lastName
public void	`setName(String, String, String)` Sets the value of the OPTIONAL customer name: firstName, middleInitial, lastName.
public void	`setPaymentDesignator(String)` Sets the Payment Designator as part of the AUTHORIZE REQUIRED payment information.
public void	`setPaymentType(String)` Sets the Payment Type as part of the AUTHORIZE and CREDIT REQUIRED payment information.
public void	`setStateOrProvince(String)` Sets the value of OPTIONAL state or province.
public void	`setTelephone(String)` Sets the values of OPTIONAL telephone number.
public void	`setToken(String)` Sets the value of the token.
public void	`setZipCode(String)` Sets the value of OPTIONAL zipcode.

Inherited Member Summary

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructors

CardHolderData()

```
public CardHolderData()
```

Constructs a CardHolderData object that does not log.

CardHolderData(String, String)

```
public CardHolderData(String sessionLoggingID
                      String eccoPropertiesFile)
```

Constructs a CardHolderData object that logs using the specified ID and properties file.

Methods

clearAllFields()

```
public void clearAllFields()
```

Clears all fields in the current CardHolderData object.

clearCardholderData)

```
public void clearCardholderData()
```

Clears the cardholder data fields, which include account, name and address information. The token field is unchanged.

clearToken()

```
public void clearToken()
```

Clears the token field, leaves the cardholder data fields unchanged.

copyValueOf(CardHolderData)

```
public void copyValueOf(CardHolderData copySourceCardHolderData)
```

Copies all values from the source CardHolderData to this CardHolderData

Parameters:
`copySourceCardHolderData` - Source for copy

getABANumber()

```
public String getABANumber()
```

Returns the ABA Number for this transaction or null if a ABA Number is not part of the transaction’s Account. Maximum size is 9.

getAccountNumber()

```
public String getAccountNumber()
```

Returns the Account Number for this transaction. Maximum size is 48.

getAddressLine1()

```
public String getAddressLine1()
```

Returns address line 1 for this transaction or null if not available. Maximum size is 100.

getAddressLine2()

```
public String getAddressLine2()
```

Returns address line 2 for this transaction or null if not available. Maximum size is 100.

getAuthorizationString()

```
public String getAuthorizationString()
```

Returns the authorization code and date of authorization of this transaction. The date of authorization is year-month-day in the form YYYY-MM-DD. Maximum size for the authorization code is 10.

getCaptureDate()

```
public String getCaptureDate()
```

Returns the capture date of the transaction as YYYY-MM-DD or null if not available.

getCity()

```
public String getCity()
```

Returns the city for this transaction or null if not available. Maximum size is 50.

getExpirationDate()

```
public String getExpirationDate()
```

Returns the expiration date as MMYY or null if the not available.

getExpirationMonth()

```
public String getExpirationMonth()
```

Returns the 2-digit expiration month for this transaction or null if an Expiration Date is not part of the transaction's Account. Values are 01-12.

getExpirationYear()

```
public String getExpirationYear()
```

Returns the 2-digit expiration year for this transaction or null if an Expiration Date is not part of the transaction's Account.

getFirstName()

```
public String getFirstName()
```

Returns the first name or null if not available. Maximum size is 60.

getLastNameOrCompanyName()

```
public String getLastNameOrCompanyName()
```

Returns the last or company name or null if not available. Maximum size is 60.

getMiddleInitial()

```
public String getMiddleInitial()
```

Returns the middle initial of the name or null if not available. Maximum size is 2.

getName()

```
public String getName()
```

Returns the name for this transaction or null if not available. Name is FirstName MiddleInitial LastName. Maximum size is 186.

getPaymentDesignator()

```
public String getPaymentDesignator()
```

Returns the payment designator for this transaction or null if a payment designator is not part of the transaction's Account. Values are ECCOClient.CHECK_DESIGNATOR_CORPORATE or ECCOClient.CHECK_DESIGNATOR_CONSUMER.

getPaymentType()

```
public String getPaymentType()
```

Returns the payment type for this transaction. Maximum size is 30. ECCOClient - PAY_AMERICAN_EXPRESS,PAY_CARTE_BLANCHE, PAY_CASH, PAY_CHECK_DRAFT, PAY_DINERS_CLUB, PAY_DIRECT_DEBIT, PAY_DISCOVER, PAY_MASTER_CARD, PAY_PAPER_CHECK, PAY_VISA

getStateOrProvince()

```
public String getStateOrProvince()
```

Returns the customer address state or province for this transaction or null if not available. Maximum size is 2.

getTelephoneNumber()

```
public String getTelephoneNumber()
```

Returns the telephone number for this transaction or null if not available. Maximum size is 16.

getToken()

```
public String getToken()
```

Returns the nine character token for this trancaction, or null if there is no token.

getZipCode()

```
public String getZipCode()
```

Returns zipcode for this transaction or null if not available.

setABANumber(String)

```
public void setABANumber(String aBANumber)
```

Sets the ABA Number as part of the REQUIRED payment information for an authorize request. When ABA Number is set individually, the other related fields must also be set: AccountNumber, PaymentDesignator, PaymentType. ABA Number is used with Payment Types PAY_DIRECT_DEBIT and PAY_CHECK_DRAFT. Maximum size is 9.

Parameters:

`aBANumber` - The bank routing number for a checking account.

setAccountNumber(String)

```
public void setAccountNumber(String accountNumber)
```

Sets the Account Number as part of the REQUIRED payment information for an authorize, credit, or record only request. When payment type is set individually, the other related fields must also be set. When Payment type is PAY_CREDIT_CARD or a specific card type (PAY_AMERICAN_EXPRESS, PAY_CARTE_BLANCHE, PAY_DINERS_CLUB, PAY_DISCOVER, PAY_MASTER_CARD, PAY_VISA), Payment Type and ExpirationDate must also be set for authorize or credit requests; record only requests do not require ExpirationDate. When Payment Type is PAY_DIRECT_DEBIT or PAY_CHECK_DRAFT, Payment Type, ABANumber and PaymentDesignator must also be set. Maximum size is 48.

Parameters:

`accountNumber` - For credit cards, the credit card number, for direct debit and check draft, the account number.

setAddress(String, String, String)

```
public void setAddress(String city,
                       String stateOrProvince,
                       String zipCode)
```

Sets the OPTIONAL address values for city, state or province and zipcode.

Parameters:

`city` - The customer city. Maximum size is 50.

`stateOrProvince` - The state or province. Maximum size is 2.

`zipCode` - The zipCode or postal code. Maximum size is 12. Only alphas and numerics permitted.

setAddress(String, String, String, String)

```
public void setAddress(String addressLine1,
               String city,
               String stateOrProvince,
               String zipCode)
```

Sets the OPTIONAL address values for address line one, city, state or province and zipcode.

Parameters:

`addressLine1` - The first line of the customer’s address. Maximum size is 100.

`city` - The customer city. Maximum size is 50.

`stateOrProvince` - The state or province. Maximum size is 2.

`zipCode` - The zipCode or postal code. Maximum size is 12. Only alphas and numerics permitted.

set_AddressLine1(String)

```
public void setAddressLine1(String addressLine1)
```

Sets the value of OPTIONAL address line one. Maximum size is 100.

Parameters:

`addressLine1` - The first line of the customer’s address. Maximum size is 100.

set_AddressLine2(String)

```
public void setAddressLine2(String addressLine2)
```

Sets the value of OPTIONAL address line two. Maximum size is 100.

Parameters:

`addressLine2` - The second line of the customer’s address. Maximum size is 100.

setCheckDraft(String, String, String)

```
public void setCheckDraft(String accountNumber,
               String aBANumber,
               String paymentDesignator)
```

Sets all REQUIRED check draft payment information for an authorize or credit request.

Parameters:

`accountNumber` - The check number for the check draft. Maximum size is 48.

`aBANumber` - The bank routing number for the check draft. Maximum size is 9.

`paymentDesignator` - The checking account type which must be CHECK_DESIGNATOR_CORPORATE or CHECK_DESIGNATOR_CONSUMER

setCity(String)

```
public void setCity(String city)
```

Sets the value of OPTIONAL city. Maximum size is 50.

Parameters:

`city` - The customer city. Maximum size is 50.

setCreditCard(String, String, String)

```
public void setCreditCard(String accountNumber,
               String expirationMonth,
               String expirationYear)
```

Sets all REQUIRED credit card payment information for an authorize or credit request.

Parameters:

`accountNumber` - The account number for the credit card. Maximum size is 48.

`expirationMonth` - The expiration month as two digits: nn (01 to 12)

`expirationYear` - The expiration year as two digits: nn

setCreditCard(String, String, String, String)

```
public void setCreditCard(String accountNumber,
               String expirationMonth,
               String expirationYear,
               String paymentType)
```

Sets all REQUIRED credit card payment information for an authorize or credit request. Also sets OPTIONAL payment type.

Parameters:

`accountNumber` - The account number for the credit card. Maximum size is 48.

`expirationMonth` - The expiration month as two digits: nn (01 to 12)

`expirationYear` - The expiration year as two digits: nn

`paymentType` - User selected payment type for cross validation with account number. ECCOClient payment types: PAY_AMERICAN_EXPRESS, PAY_CARTE_BLANCHE PAY_DINERS_CLUB, PAY_DISCOVER, PAY_MASTER_CARD, PAY_VISA

setDirectDebit(String, String, String)

```
public void setDirectDebit(String accountNumber,
               String aBANumber,
               String paymentDesignator)
```

Sets all REQUIRED direct debit payment information for an authorize or credit request.

Parameters:

`accountNumber` - The account number for direct debit. Maximum size is 48.

`aBANumber` - The bank routing number for direct debit. Maximum size is 9.

`paymentDesignator` - The checking account type which must be CHECK_DESIGNATOR_CORPORATE or CHECK_DESIGNATOR_CONSUMER

setExpirationDate(String, String)

```
public void setExpirationDate(String expirationMonth,
               String expirationYear)
```

Sets the Expiration Date as part of the AUTHORIZE and CREDIT REQUIRED payment information. When the Expiration Date is set individually, the other related fields must also be set: AccountNumber, PaymentType. Expiration Date is used with Payment Type PAY_CREDIT_CARD and the specific card types: PAY_AMERICAN_EXPRESS, PAY_CARTE_BLANCHE, PAY_DINERS_CLUB, PAY_DISCOVER, PAY_MASTER_CARD, PAY_VISA.

Parameters:

`expirationMonth` - The expiration month as two digits: nn (01 to 12)

`expirationYear` - The expiration year as two digits: nn

setExpirationMonth(String)

```
public void setExpirationMonth(String expirationMonth)
```

Sets the Expiration Month as part of the REQUIRED payment information for an authorize or credit request. When the Expiration Month is set individually, the other related fields must also be set: Expiration Year, AccountNumber, PaymentType. Expiration Month is used with Payment Type PAY_CREDIT_CARD and the specific card types: PAY_AMERICAN_EXPRESS, PAY_CARTE_BLANCHE, PAY_DINERS_CLUB, PAY_DISCOVER, PAY_MASTER_CARD, PAY_VISA.

Parameters:

`expirationMonth` - The expiration month as two digits: nn (1 to 12)

setExpirationYear(String)

```
public void setExpirationYear(String expirationYear)
```

Sets the Expiration Year as part of the REQUIRED payment information for an authorize or credit request. When the Expiration Year is set individually, the other related fields must also be set: Expiration Month, AccountNumber, PaymentType. Expiration Year is used with Payment Type PAY_CREDIT_CARD and the specific card types: PAY_AMERICAN_EXPRESS, PAY_CARTE_BLANCHE, PAY_DINERS_CLUB, PAY_DISCOVER, PAY_MASTER_CARD, PAY_VISA.

Parameters:

`expirationYear` - The expiration year as two digits: nn

setName(String)

```
public void setName(String lastNameOrCompany)
```

Sets the value of the OPTIONAL customer name to either the company name or last name. Maximum size is 60.

Parameters:

`lastNameorCompany` - Last name of customer or company name.

setName(String, String)

```
public void setName(String firstName,
               String lastName)
```

Sets the value of the OPTIONAL customer name: firstName, lastName

Parameters:

`firstName` - The first name of the customer. Maximum size is 60.

`lastName` - The last name of the customer. Maximum size is 60.

setName(String, String, String)

```
public void setName(String firstName,
               String middleInitial,
               String lastName)
```

Sets the value of the OPTIONAL customer name: firstName, middleInitial, lastName.

Parameters:

`firstName` - The first name of the customer. Maximum size is 60.

`middleInitial` - The middle initial of the customer. Maximum size is 2.

`lastName` - The last name of the customer. Maximum size is 60.

setPaymentDesignator(String)

```
public void setPaymentDesignator(String paymentDesignator)
```

Sets the Payment Designator as part of the REQUIRED payment information for an authorize or credit request for a direct debit or checking account. When Payment Designator is set individually, the other related fields must also be set: AccountNumber, ABANumber, PaymentType. Payment Designator is used with Payment Types PAY_DIRECT_DEBIT and PAY_CHECK_DRAFT.

Parameters:

`paymentDesignator` - The checking account type which must be CHECK_DESIGNATOR_CORPORATE or CHECK_DESIGNATOR_CONSUMER

setPaymentType(String)

```
public void setPaymentType(String paymentType)
```

Sets the Payment Type as part of the REQUIRED payment information for an authorize or credit request. When payment type is set individually, the other related fields must also be set. When Payment type is PAY_CREDIT_CARD or a specific card type (PAY_AMERICAN_EXPRESS, PAY_CARTE_BLANCHE, PAY_DINERS_CLUB, PAY_DISCOVER, PAY_MASTER_CARD, PAY_VISA), AccountNumber and ExpirationDate must also be set. When Payment Type is PAY_DIRECT_DEBIT or PAY_CHECK_DRAFT, AccountNumber, ABANumber and PaymentDesignator must also be set.

Parameters:

`paymentType` - Payment type which must be one of the following:

PAY_CREDIT_CARD,PAY_AMERICAN_EXPRESS, PAY_CARTE_BLANCHE, PAY_DINERS_CLUB, PAY_DISCOVER, PAY_MASTER_CARD, PAY_VISA, PAY_DIRECT_DEBIT, or PAY_CHECK_DRAFT.

setStateOrProvince(String)

`public void setStateOrProvince(String stateOrProvince)`

Sets the value of OPTIONAL state or province code. Maximum size is 2.

Parameters:
`stateOrProvince` - Two-character state or province code.

setTelephone(String)

`public void setTelephone(String phoneNumber)`

Sets the values of OPTIONAL telephone number. Maximum size is 16. No formatting is required.

Parameters:
`phoneNumber` - Unformatted phone number string. Maximum size is 16.

setToken(String)

`public void setToken(String token)`

Sets the value of the token field. Tokens are nine character strings representing cardholder data.

Parameters:
`token` - nine character token or empty string.

setZipCode(String)

`public void setZipCode(String zipCode)`

Sets the value of OPTIONAL zipcode. Note, however, that zip code is required for address verification. Only alphanumeric characters are permitted. Maximum size is 12.

Parameters:
`zipCode` - Zip code or postal code, expressed as alphanumeric string. Maximum size is 12.

TransactionSelection

Declaration

```
public class TransactionSelection

java.lang.Object
  |
  +--com.edgil.ecco.eccoapi.TransactionSelection
```

Description

A TransactionSelection object contains the selection criteria used to retrieve transaction information from the EdgCapture database. Methods marked with ** are exclusively used for retrieval of multiple transactions..

Member Summary	
Constructors	
`public`	`TransactionSelection()`
`public`	`TransactionSelection(String, String)`
Methods	
`public void`	`clearAllEntries()` Clear all previously set selection criteria.
`public void`	`clearAllEntriesExceptMerchant()` Clears all previously set selection criteria except the Merchant(s) and their Oeps.
`public String`	`**getECCOContextForPaging()` Gets the ECCO context string holding the directory ID for paging.
`public MonetaryTransactionData`	`**getNextTransaction()` Gets the next MonetaryTransactionData record from the set of rows returned from EdgCapture.
`public int`	`**getNumberRowsFound()` Gets the number of rows found for this directory selection.
`public int`	`**getNumberRowsReturned()` Gets the number of rows returned for this directory.
`public MonetaryTransactionData`	`**getTransactionAtIndex(int)` Gets the MonetaryTransactionData record at the specified index from the set of rows returned from EdgCapture.
`void`	`**resetIndex()` Resets the enumeration through the MonetaryTransactionData rows back to the beginning.
`public void`	`**setNumberRowsRequested(int)` Sets the value for the maximum number of rows to be returned at one time.
`public void`	`**setProcessSource(String)` Sets process source to limit selection.
`public void`	`**setSelectionAccountNumber(String)` Sets the value of the Account Number to limit selection.

Member Summary	
public void	`**setSelectionBeginDate(String, String, String)` Sets the begin date for Date Range selection.
public void	`**setSelectionClientUserName(String)` Sets one or more client user logon names to limit selection.
public void	`**setSelectionClientUserName(String[])` Sets one or more client user logon names to limit selection.
public void	`**setSelectionDateOption(int)` Sets a date option for selection.
public void	`**setSelectionEndDate(String, String, String)` Sets the end date for Date Range selection.
public void	`**setSelectionLastNameOrCompanyName(String)` Sets the value of the Customer Last name or Company name to limit selection.
public void	`**setSelectionMerchantAndOep(String, String)` Sets a Merchant and one of its OEPs for selection.
public void	`**setSelectionPaging(String)` Continue paging by giving EdgCapture the ECCO context string identifying rows held from a previous selection.
public void	`**setSelectionPaymentType(String)` Sets a payment type to limit selection.
public void	`**setSelectionPaymentType(String[])` Sets one or more payment type to limit selection.
public void	`**setSelectionQueueName(String)` Sets a queue name to limit selection.
public void	`**setSelectionQueueName(String[])` Sets one or more queue name to limit selection.
public void	`**setSelectionTelephoneNumber(String)` Sets the value of the telephone number to limit selection.
public void	`**setSelectionToken(String)` Sets the value of the token. Only transactions for the related cardholder.
public void	`setSelectionTransactionKey(String, String, String)` Sets the values for the three fields REQUIRED to select a unique transaction: Merchant, OEP, TransactionId.
public void	`setSelectionTransactionKeyMerchant(String)` Sets the value of the REQUIRED MerchantId for selecting a unique transaction using the Key: Merchant, OEP, TransactionId.
public void	`setSelectionTransactionKeyOep(String)` Sets the value of the REQUIRED OEPId for selecting a unique transaction using the Key: Merchant, OEP, TransactionId.
public void	`setSelectionTransactionKeyTransactionId(String)` Sets the value of the REQUIRED Transaction Id for selecting a unique transaction using the Key: Merchant, OEP, TransactionId.
public void	`**setSelectionTransactionType(String)` Sets a transaction type to limit selection.
public void	`**setSelectionTransactionType(String[])` Sets one or more transaction types to limit selection.

Member Summary	
`public void`	`**setSelectionUserData1(String)` Sets the value of the 40 character UserData1 field to limit selection.
`public void`	`**setSelectionUserData10(String)` Sets the value of the 40 character UserData10 field to limit selection.
`public void`	`**setSelectionUserData2(String)` Sets the value of the 40 character UserData2 field to limit selection.
`public void`	`**setSelectionUserData3(String)` Sets the value of the 40 character userData3 field to limit selection.
`public void`	`**setSelectionUserData4(String)` Sets the value of the 40 character UserData4 field to limit selection.
`public void`	`**setSelectionUserData5(String)` Sets the value of the 40 character UserData5 field to limit selection.
`public void`	`**setSelectionUserData6(String)` Sets the value of the 40 character UserData6 field to limit selection.
`public void`	`**setSelectionUserData7(String)` Sets the value of the 40 character UserData7 field to limit selection.
`public void`	`**setSelectionUserData8(String)` Sets the value of the 40 character UserData8 field to limit selection.
`public void`	`**setSelectionUserData9(String)` Sets the value of the 40 character UserData9 field to limit selection.

Inherited Member Summary
Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructors

TransactionSelection()

`public **TransactionSelection**()`

Methods

clearAllEntries()

`public void **clearAllEntries**()`

Clears all previously set selection criteria.

setSelectionToken(String)

```
public void setSelectionToken(String token)
```

Sets the value for the token selection.

Parameters:

`token` - Nine character string representing a token, or empty string meaning no token selection.

clearAllEntriesExceptMerchant()

```
public void clearAllEntriesExceptMerchant()
```

Clears all previously set selection criteria except the Merchant(s) and their OEPs.

setSelectionTransactionKey(String, String, String)

```
public void setSelectionTransactionKey(String merchantId,
          String oepId,
          String transactionId)
```

Sets the values for the three fields REQUIRED to select a unique transaction: Merchant, OEP, TransactionId.

Parameters:

`merchantId` - Fixed string identifying the destination bank account to EdgCapture.

`oepId` - Fixed string identifying the client subaccount to EdgCapture.

`transactionId` - String that uniquely identifies the transaction within the Merchant and the OEP.

setSelectionTransactionKeyMerchant(String)

```
public void setSelectionTransactionKeyMerchant(String merchantId)
```

Sets the value of the REQUIRED MerchantId for selecting a unique transaction using the Key: Merchant, OEP, TransactionId. The MerchantId is a fixed string identifying the destination bank account to EdgCapture.

Parameters:

`merchantId` - Fixed string identifying the destination bank account to EdgCapture.

setSelectionTransactionKeyOep(String)

```
public void setSelectionTransactionKeyOep(String oepId)
```

Sets the value of the REQUIRED OEPId for selecting a unique transaction using the Key: Merchant, OEP, TransactionId.

Parameters:

`oepId` - Fixed string identifying the client subaccount to EdgCapture.

setSelectionTransactionKeyTransactionId(String)

```
public void setSelectionTransactionKeyTransactionId(String
          transactionId)
```

Sets the value of the REQUIRED Transaction Id for selecting a unique transaction using the Key: Merchant, OEP, TransactionId.

Parameters:

`transactionId` - String that uniquely identifies the transaction within the Merchant and the OEP.

Methods available for multiple transaction retrieval

getNextTransaction()

```
public MonetaryTransactionData getNextTransaction()
```

Gets the next MonetaryTransactionData record from the set of rows returned from EdgCapture.

getNumberRowsFound()

```
public int getNumberRowsFound()
```

Gets the number of rows found for this directory selection.

getNumberRowsReturned()

```
public int getNumberRowsReturned()
```

Gets the number of rows returned for this directory.

setNumberRowsRequested(int)

```
public void setNumberRowsRequested(int iRows)
```

Sets the value for the maximum number of rows to be returned at one time.

Parameters:

`iRows` - The maximum number of rows to return.

getECCOContextForPaging(String)

```
public String getECCOContextForPaging()
```

Gets the ECCO Context String holding the Directory ID for paging. When more rows are found than requested, EdgCapture holds the surplus rows and returns a Context with a Directory specific id to reclaim them. Paging moves forward through the rows represented by the ECCO Context.

getTransactionAtIndex(int)

```
public MonetaryTransactionData getTransactionAtIndex(int iIndex)
```

Gets the MonetaryTransactionData record at the specified index from the set of rows returned from EdgCapture.

Parameters:

`iIndex` - The index number. Index begins at 0.

resetIndex(int)

```
void resetIndex()
```

Reset the enumeration through the MonetaryTransactionData rows back to the beginning. It is automatically set to the first row when the message returns from the EdgCapture with the rows found.

setProcessSource(String)

```
public void setProcessSource(String processSource)
```

Sets process source to limit selection. Process source is a client specified string used to tie together transactions from one or more points of origin or one or more users. Process source optionally may be specified when logging onto the EdgCapture from the ECCO API. Then it may be used to group together transactions sharing a process source into a common directory selection.

Parameters:

`processSource` - The process source for the transactions.

setSelectionAccountNumber(String)

```
public void setSelectionAccountNumber(String accountNumber)
```

Sets the value of the Account Number to limit selection. Only one account number may be used for selection.

Parameters:

`accountNumber` - The account number for the transactions.

setSelectionBeginDate(String, String, String)

```
public void setSelectionBeginDate(String month,
               String day,
               String year)
```

Sets the begin date for Date Range selection. Setting a begin date sets the date option to DATE_OP_DATERANGE.

Parameters:

`month` - Month of the year (01 - 12)

`day` - Day of the month (01 - 31)

`year` - Four digit year

TRUE if month, day and year are within range.

setSelectionClientUserName(String)

```
public void setSelectionClientUserName(String clientUserName)
```

Sets one user logon name to limit selection.

setSelectionClientUserName(String[])

```
public void setSelectionClientUserName(String[] clientUserName)
```

Sets one or more client user logon names to limit selection.

setSelectionDateOption(int)

```
public void setSelectionDateOption(int dateOption)
```

Sets a date option for selection. The date options are: DATE_OP_TODAY, DATE_OP_YESTERDAY, DATE_OP_THISWEEK, DATE_OP_LASTWEEK, DATE_OP_DATERANGE

Parameters:

`dateOption` - Date option as available from ECCOClient.

setSelectionEndDate(String, String, String)

```
public void setSelectionEndDate(String month,
              String day,
              String year)
```

Sets the end date for Date Range selection. Setting an end date sets the date option to DATE_OP_DATERANGE.

Parameters:

`month` - Month of the year (01 - 12)

`day` - Day of the month (01 - 31)

`year` - Four digit year

setSelectionLastNameOrCompanyName(String)

```
public void setSelectionLastNameOrCompanyName(String lastNameOrCompany)
```

Sets the value of the Customer Last name or Company name to limit selection. Only one name may be used to limit selection.

setSelectionMerchantAndOep(String, String)

```
public void setSelectionMerchantAndOep(String merchantId,
              String oepId)
```

Set a Merchant and one of its OEPs for selection. If OEP is empty, then all oeps for the merchant are selected. Multiple Merchant/OEP selections can be made by calling this method. Neither parameter can be null.

Note: Each call is OR'd together with previous calls

Examples:

setSelectionMerchantAndOEP ('0', 'A')

Results in the selection: Merchant = '0' AND OEP = 'A'

setSelectionMerchantAndOEP ('0', '')

Results in the selection: Merchant = '0' setSelectionMerchantAndOEP ('0', 'A')

setSelectionMerchantAndOEP ('1', 'B')

Results int he selection: (Merchant = '0' AND OEP = 'A') OR (Merchant = '1' AND OEP = 'B')

setSelectionPaging(String)

```
public void setSelectionPaging(String contextString)
```

Continue paging by giving EdgCapture the ECCO Context String identifying rows held from a previous selection. When more rows are found than requested, EdgCapture holds the surplus rows and returns a Context with a Directory specific id to reclaim them. Paging moves forward through the rows represented by the ECCO Context.

Parameters:

`contextString` - The context string containing the directory ID returned from the *`getECCOContextForPaging`* method.

setSelectionPaymentType(String)

```
public void setSelectionPaymentType(String paymentType)
```

Sets a payment type to limit selection. The payment types from the ECCOClient are: PAY_AMERICAN_EXPRESS, PAY_CARTE_BLANCHE, PAY_CASH, PAY_CHECK_DRAFT, PAY_DINERS_CLUB, PAY_DIRECT_DEBIT PAY_DISCOVER,PAY_MASTER_CARD, PAY_PAPER_CHECK, PAY_VISA.

setSelectionPaymentType(String[])

```
public void setSelectionPaymentType(String[] paymentType)
```

Sets one or more payment type to limit selection. The payment types from the ECCOClient are: PAY_AMERICAN_EXPRESS, PAY_CARTE_BLANCHE, PAY_CASH, PAY_CHECK_DRAFT, PAY_DINERS_CLUB, PAY_DIRECT_DEBIT PAY_DISCOVER,PAY_MASTER_CARD, PAY_PAPER_CHECK, PAY_VISA

setSelectionQueueName(String)

```
public void setSelectionQueueName(String queueName)
```

Sets a queue name to limit selection. The queue names from ECCOClient are: QUEUE_AUTHORIZED, QUEUE_CAPTURED, QUEUE_DECLINED, QUEUE_ERROR, QUEUE_HELD, QUEUE_INPUT, QUEUE_MARKED_FOR_CAPTURE, QUEUE_RECORDED_ONLY, QUEUE_UNKNOWN_DISPOSITION, QUEUE_VOIDED.

setSelectionQueueName(String[])

```
public void setSelectionQueueName(String[] queueName)
```

Sets one or more queue name to limit selection. The queue names from ECCOClient are: QUEUE_AUTHORIZED, QUEUE_CAPTURED, QUEUE_DECLINED, QUEUE_ERROR, QUEUE_HELD, QUEUE_INPUT, QUEUE_MARKED_FOR_CAPTURE, QUEUE_RECORDED_ONLY, QUEUE_UNKNOWN_DISPOSITION, QUEUE_VOIDED.

setSelectionTelephoneNumber(String)

`public void` **`setSelectionTelephoneNumber`**`(String phoneNumber)`

Sets the value of the telephone number to limit selection. Only one phone number may be used for selection.

setSelectionToken(String)

`public void` **`setSelectionToken`**`(String token)`

Sets the value for the token selection.

Parameters:

`token` - Nine character string respesenting a token, or empty string meaning no token selection.

setSelectionTransactionType(String)

`public void` **`setSelectionTransactionType`**`(String transactionType)`

Sets a transaction type to limit selection. The transaction types from ECCOClient are: TRAN_AUTHORIZE, TRAN_CAPTURE_HELD_TRANSACTION, TRAN_CREDIT, TRAN_DECLINE_HELD_TRANSACTION, TRAN_MARKED_FOR_CAPTURE, TRAN_RECORD_ONLY, TRAN_VOID.

Parameters:

`transactionId` - Fixed string from ECCOClient that identifies the transaction type.

setSelectionTransactionType(String[])

`public void` **`setSelectionTransactionType`**`(String[] transactionType)`

Sets one or more transaction types to limit selection. The transaction types are: TRAN_AUTHORIZE, TRAN_CAPTURE_HELD_TRANSACTION, TRAN_CREDIT, TRAN_DECLINE_HELD_TRANSACTION, TRAN_MARKED_FOR_CAPTURE, TRAN_RECORD_ONLY, TRAN_VOID.

Parameters:

`transactionId` - Fixed string from ECCOClient that identifies the transaction type.

setSelectionUserData1(String)

`public void` **`setSelectionUserData1`**`(String userData)`

Sets the value of the 40 character UserData1 field to limit selection. Only one UserData1 may be used to limit selection.

Parameters:

`userData` - Up to 40 characters of user-specified data.

setSelectionUserData10(String)

`public void **setSelectionUserData10**(String userData)`

Sets the value of the 40 character UserData10 field to limit selection. Only one UserData10 may be used to limit selection.

Parameters:
`userData` - Up to 40 characters of user-specified data.

setSelectionUserData2(String)

`public void **setSelectionUserData2**(String userData)`

Sets the value of the 40 character UserData2 field to limit selection. Only one UserData2 may be used to limit selection.

Parameters:
`userData` - Up to 40 characters of user-specified data.

setSelectionUserData3(String)

`public void **setSelectionUserData3**(String userData)`

Sets the value of the 40 character userData3 field to limit selection. Only one userData3 may be used to limit selection.

Parameters:
`userData` - Up to 40 characters of user-specified data.

setSelectionUserData4(String)

`public void **setSelectionUserData4**(String userData)`

Sets the value of the 40 character UserData4 field to limit selection. Only one UserData4 may be used to limit selection.

Parameters:
`userData` - Up to 40 characters of user-specified data.

setSelectionUserData5(String)

`public void **setSelectionUserData5**(String userData)`

Sets the value of the 40 character UserData5 field to limit selection. Only one UserData5 may be used to limit selection.

Parameters:
`userData` - Up to 40 characters of user-specified data.

setSelectionUserData6(String)

`public void **setSelectionUserData6**(String userData)`

Sets the value of the 40 character UserData6 field to limit selection. Only one UserData6 may be used to limit selection.

Parameters:
`userData` - Up to 40 characters of user-specified data.

setSelectionUserData7(String)

`public void` **`setSelectionUserData7`**`(String userData)`

Sets the value of the 40 character UserData7 field to limit selection. Only one UserData7 may be used to limit selection.

Parameters:
`userData` - Up to 40 characters of user-specified data.

setSelectionUserData8(String)

`public void` **`setSelectionUserData8`**`(String userData)`

Sets the value of the 40 character UserData8 field to limit selection. Only one UserData8 may be used to limit selection.

Parameters:
`userData` - Up to 40 characters of user-specified data.

setSelectionUserData9(String)

`public void` **`setSelectionUserData9`**`(String userData)`

Sets the value of the 40 character UserData9 field to limit selection. Only one UserData9 may be used to limit selection.

Parameters:
`userData` - Up to 40 characters of user-specified data.

The ECCO C++ API

This section describes the ECCO C++ API. It includes short code samples illustrating basic functionality and complete class reference information.

Using the C++ API

This section presents more detailed examples of how to process transactions and retrieve information from EdgCapture using the C++ wrapper for the Java API.

Setting up the ECCO.properties file

The ECCO API is delivered with a properties file that is set to use a “Fake” connection to EdgCapture. In “Fake” mode, messages are never actually transmitted to the EdgCapture server. All responses are generated by the java API itself.

When you are ready to submit a transaction to EdgCapture, you must edit the ECCO.properties file to change the connection type to “Ssl” and to specify the connection information for the EdgCapture server. Note that this parameter is case-sensitive and must be entered as documented here with an initial upper-case character.

See “Appendix D: Editing Properties Files” on page 199 for a complete description of the Properties file and the connection information for the EdgCapture server.

The ECCO.properties file also controls logging from the API. For information on logging, see “Logging during development” on page 25.

Processing a transaction

The procedure here covers a complete sale transaction, consisting of an authorization request and a mark for capture request. It also explains how to obtain a unique transaction identifier from EdgCapture for an application that does not produce its own identifiers.

Certification

The secure connection between the application and the EdgCapture server is established at startup of the application and before any transactions are submitted.

The ECCOClient method CertifyECCO currently uses edgilca.keystore as described in “Appendix D: Editing Properties Files” on page 199.

Create an ECCOClient object and call the ECCOClient method *certifyECCO* passing the required passwords and certificate alias to certify the client and server. Edgil recommends that the passwords be entered at startup by an actual user, instead of reading them from a configuration file, so that they are not stored as clear text anywhere on the system.

The first parameter for CertifyECCO is the file password for edgilca.keystore, the others are placeholders for future development.

```
ECCOClient *pxClient = new ECCOClient();

int status = pxClient->CertifyECCO(argv[0], argv[1], argv[2]);
```

The default values for edgilca.keystore can be used.

Argv[0]	ChangeIt	<case sensitive>	certificate pwd
Argv[1]	ChangeIt	<case sensitive>	certificate pwd
Argv[2]	edgilca	<case sensitive>	certificate alias

Creating the connection

In order to submit a transaction or an information request to EdgCapture, the application must establish a connection to the server by creating an ECCO client. When you create the ECCOClient object, you can also set up client-side logging for debugging purposes, as described in “Logging during development” on page 25.

Create the ECCOClient object.

```
#include "ECCOClient.h"

ECCOClient *pxClient = new ECCOClient();
```

Call the ECCOClient method *Logon,* passing logon and password as parameters for manual processing. Note that the logon and password belong to the application and can be used for any connection. The logon method opens a secure socket, certifying client and server, and sends the logon message to EdgCapture.

```
// Log onto the EdgCapture Server specifying manual processing
int status = pxClient->Logon("myLogon","myPassword");
```

Test the results of the request. There are a few reasons for failure, including invalid logon or password, security errors, or connection problems.

Note: Edgil will provide the credentials for ECCOClient method logon to the EdgCapture server.

Default values are:

```
int status = pxClient->Logon("EccoClient","EccoClient1");
```

Creating and populating the monetary transaction data

Create a *MonetaryTransactionData* object.

```
#include "MonetaryTransactionData.h"

MonetaryTransactionData *pxTransactionData =
   new MonetaryTransactionData();
```

Set the data fields in the *MonetaryTransactionData* object. MonetaryTransactionData provides a set method for each individual data field, as well as methods to set several fields at once.

EdgCapture requires a unique transaction identifier for each transaction within a Merchant and OEP. The required fields for an authorization or credit request include:

Data	*Method*	*Description*
MerchantId	SetMerchantId	Merchant identifier
OEPId	SetOEPId	OEP identifier
Transaction Id	SetTransactionId	Unique transaction identifier
Amount	SetAmount	Amount of the sale or credit in format *nnnn.nn*; must be positive for all transaction types.
Payment information	setCreditCard setCheckDraft setDirectDebit setToken	Required credit card data includes AccountNumber and ExpirationDate. Required check draft and direct debit data includes ABANumber, AccountNumber, and PaymentDesignator. Token requires a valid CDM token
ECI	SetECI	Electronic commerce indicator that describes the source of the transaction. For web applications, the value is generally 6, indicating that the customer’s account data was transmitted using SSL.

See “Appendix C: Monetary Transaction Data” on page 189 for a complete description of the data fields. All data fields should be specified as strings.

The following example assumes that the application has created its own transaction id. It sets some data fields individually and others in logical groups:

Note: Actual setMerchantId and setOEPId values are determined by Edgil Associates.

```
pxTransactionData->SetMerchantId("0");
pxTransactionData->SetOEPId("A");
pxTransactionData->SetTransactionId("mN23d88");
pxTransactionData->SetAmount("95.40");
pxTransactionData->SetAccountNumber("9999999999999999");
pxTransactionData->SetExpirationDate("10","12");
pxTransactionData->SetECI("6");
pxTransactionData->SetFraudSecurityValue("123");
pxTransactionData->SetAddress("99 Wayward Lane","Boston", "MA",
   "01776");
pxTransactionData->SetName("Jane","Smith");
pxTransactionData->SetUserData1("Classified");
pxTransactionData->SetUserData2("auto");
pxTransactionData->SetUserData3("daily morning,weekly shopper");
pxTransactionData->SetUserData4("by Alice");
pxTransactionData->SetUserData6("Jan 5 2010");
```

Submitting the transaction to EdgCapture

Call the ECCOClient method *RequestAuthorization* passing the previously created MonetaryTransactionData as a parameter.

```
int status = pxClient->RequestAuthorization(*pxTransactionData);
```

Processing the results

Test the return status from *RequestAuthorization* to determine whether the transaction request has succeeded. The MonetaryTransactionData object will have all the information returned from EdgCapture, including the authorization code and other processing information for a successful transaction. If the request fails, the fields in the MonetaryTransactionData object remain as they were set before sending the request. If EdgCapture has recorded the transaction, it returns the transaction status, the current queue and other information.

SUCCESS

On ECCO_SUCCESS, the application should record the results, including MerchantId, OEPId, and TransactionId. For a successfully authorized transaction, the application must mark the transaction for capture to notify EdgCapture that the transaction is complete from the application side. If no mark for capture request is sent, the transaction will not be settled. See "Sending a mark for capture request" on page 107 for a description of this process.

To retrieve the authorization code, the current EdgCapture queue, or previously set transaction data, use the MonetaryTransactionData's get methods. For example:

```
char ztReceive[512];

char *pAuthorizationCode =
   pxTransactionData->GetAuthorizationCode(ztReceive);
```

FAILURE

Check the return status against the codes in ECCO.h. See "Appendix B: ECCO Status Codes" on page 189 for more information.

The type of failure determines the action needed. For data entry errors and declined credit cards, the customer may be asked to reenter information or another account number. The transaction can then be resubmitted using the same connection and Transaction Id.

```
switch status
   {
   case ECCO_COMPROBLEM:
      //tell the customer to come back later
      break;
   case ECCO_INVALIDACCOUNTNUMBER:
      //ask the customer to check it or provide another account
      break;
   }
```

If you want to display the message text, use the ECCOClient method *GetStatusMessage* to retrieve it.

Sending a mark for capture request

The current MonetaryTransaction object can also be used to send the request to mark the transaction for capture.

After the application has ensured that the product has been shipped or delivered to the customer, the successfully authorized transaction must be marked for capture. If the MonetaryTransactionData object reporting the successful authorization still exists, it contains all the necessary information, and the application can use it for the request by calling the ECCOClient's *RequestMarkForCapture* method.

```
int status = pxClient->RequestMarkForCapture(*pxTransactionData);
```

If the application must send the mark for capture message at a later time, follow the steps in the previous section to create a new *ECCOClient* connection and a new *MonetaryTransactionData* object. Set the MerchantId, OEPId, and TransactionId to the values stored for this transaction, and call *RequestMarkForCapture*, passing the MonetaryTransactionData as a parameter.

```
   pxTransactionData->SetMerchantId("0");
   pxTransactionData->SetOEPId("A");
   pxTransactionData->SetTransactionId("mN23d88");

int status = pxClient->RequestMarkForCapture(*pxTransactionData);
```

Test the return status for success. See "Appendix B: ECCO Status Codes" on page 189 for information.

Reusing the connection for a subsequent transaction

You can reuse the same client connection for a new transaction or to resubmit a transaction after correcting errors in the data.

When the next transaction is entirely different from the current transaction, clear the MonetaryTransaction object of existing data using *ClearAllFields*, which returns all fields to null.

```
pxTransactionData->ClearAllFields();
//now set the new ones
```

When a transaction request needs to be resent because of a minor error or when a new transaction is to be performed for the same customer, you do not need to clear the existing data before resetting the incorrect fields. In this example, the customer has entered a new account number:

```
pxTransactionData->SetAccountNumber("44444444444444");
```

To continue, submit the transaction request to EdgCapture.

Cleaning up

After all messages have been sent using the current ECCOClient, clean up and disconnect by calling the ECCOClient method *Logoff*.

Retrieving transaction data for a single transaction

Each transaction is uniquely identified in the EdgCapture database by its Transaction Key, made up of the Merchant Id, the OEP Id, and the Transaction Id. To retrieve the data for any transaction, the application must have this information available. Note that you must perform a transaction selection in order to void a previously submitted transaction.

If no connection exists, create an ECCOClient and log on, as described in the previous section. Otherwise, use the existing ECCOClient connection.

Create a *TransactionSelection*.

```
#include "TransactionSelection.h"

TransactionSelection *pxSelection = new TransactionSelection();
```

Use the TransactionSelection's *SetSelectionTransactionKey* method to set the required Merchant Id, OEP Id and TransactionId that uniquely identify the transaction. You can also set the fields individually.

```
pxSelection->SetSelectionTransactionKey("0","B","BE345");
```

Call the ECCOClient method *RequestDirectory* passing the previously created TransactionSelection as a parameter.

```
int status = pxClient->RequestDirectory(*pxSelection);
```

Test the return status for success. There are few reasons to fail, although it is possible if the application requests a Merchant or OEP to which it does not have access or if the requested transaction is not in the database.

If the request is successful, access the returned MonetaryTransactionData object.

```
if (status == ECCO_SUCCESS)
{
   //pick up the data
   MonetaryTransactionData myTransactionData;

   if ((pxSelection->GetNextTransaction(myTransactionData))==
   TRUE)
   {
   //use get methods to retrieve data
   }
}
```

Use MonetaryTransactionData's get methods to access the data. In this example, the application checks the queue to determine whether the transaction has been authorized.

```
char ztReceive[512];

char *pQueue = myTransactionData->GetQueueName(ztReceive);
```

To retrieve the data for another transaction using the same ECCOClient connection, reset the selection criteria and call *RequestDirectory* again.

To close the connection, call the ECCOClient's *Logoff* method.

Voiding a transaction

A void transaction nullifies a previous sale or credit transaction that has not yet been captured. A void can only be entered on the same day during the same settlement period as the transaction it is meant to offset. Voids are used to correct a mistake, such as an incorrect amount or card number, or to cancel a transaction submitted in error.

To void a transaction, request the transaction information as described in the previous section. Using the returned MonetaryTransactionData object, send the void request.

```
int status = pxClient->RequestVoid(*pxTransactionData);
```

Retrieving a directory of transactions

To retreive a group of transactions:

Create the *TransactionSelection* object.

```
TransactionSelection *pxSelection = new TransactionSelection();
```

Specify the number of transactions to return by setting a row count. Each row corresponds to the data for a single transaction. This example requests 20 transactions:

```
#include "TransactionSelection.h"

pxSelection->SetNumberRowsRequested(20);
```

Use TransactionSelection's set methods to specify the selection criteria. The selection criteria are defined in the ECCOClient. This example requests all Visa transactions from yesterday that are in the captured queue, that is, all Visa transactions that were included in the previous night's settlement batch.

```
pxSelection->SetSelectionPaymentType(PAY_VISA);
```

```
pxSelection->SetSelectionQueue(QUEUE_CAPTURED);
pxSelection->SetSelectionDateOptions(DATE_OP_YESTERDAY);
```

Send the request to EdgCapture using the ECCOClient's *RequestDirectory* method.

```
int status = pxClient->RequestDirectory(*pxSelection);
```

If the request is successful, access the MonetaryTransactionData objects. Use *GetNumberRowsReturned* to determine how many transactions were returned. Note that the maximum number of rows for the current example is 20, as specified previously. To determine how many rows the query actually retrieved from the EdgCapture database, use TransactionSelection's *GetNumberRowsFound* method.

```
if (status == ECCO_SUCCESS)
{
   int numberOfRows;
   MonetaryTransactionData myTransactionData;

   numberOfRows = pxSelection->GetNumberRowsReturned();
   for (i = 0; i < numberOfRows; i++)
   {
      if ((pxSelection->GetNextTransaction(myTransactionData))==
            TRUE)
      {
      //use get methods to retrieve data
      }
   }
}
```

Use the MonetaryTransactionData's get methods to access the data from individual fields.

Updating User Data for Transactions

Transactions have user data associated with them. These fields are labelled User1 through User10, plus the notes field. These fields are used to store additional data in EdgCapture for reference by the system sending transactions. For example, a system may use a user field to link EdgCapture transactions to it's own order by storing the order id in a user field.

The update user data request allows the updating of all user data fields plus the notes field. The data is sent via a MonetaryTransactionData object. This object may be returned via a transaction selection, re-used from a previously submitted transaction, or generated from scratch.

RequestUpdateUserData will overwrite the existing user data for ALL of the following fields in the associated MonetaryTransactionData object: UserData1 through UserData10 and Notes. All other fields will remain unchanged.

Each transaction is uniquely identified in the EdgCapture database by its TransactionKey, made up of the Merchant Id, the OEP Id, and the Transaction Id. To update the user data for any transaction, the application must have this information available so that the transaction to be updated can be uniquely identified.

If no connection exists, create an ECCOClient and log on, as described in the previous sections. Otherwise, use the existing ECCOClient connection.

You could use a previously used MonetaryTransactionData object or create and populate one with the appropriate data in the transaction key, user fields, and notes field. Use the MonetaryTransactionData's *setTransactionKey* method to set the required MerchantId, OEPId and TransactionId that uniquely identify the transaction. Alternatively, you can also set the fields individually using the setters for TransactionId, MerchantId, and OEPId:

Many times you will be re-using a MonetaryTransactionData object that was returned from a transaction selection or used to process a transaction. In this case you only need to update the user fields and notes fields as needed.

```
int status = pxTransactionData->SetUserData1("data") ;
```

Once the data is ready, call the ECCOClient method RequestUpdateUserData:

```
int status = pxClient->RequestUpdateUserData(*pxTransactionData);
```

Test the return status for success. If the request is successful, then the specified transaction will have the user data and notes field updated according to what was set in the MonetaryTransactionData.

To close the connection, call the ECCOClient’s logoff method.

Debugging a C++ application

The C++ API is based upon the underlying java implementation. Debugging can involve either the interface between the ECCO C++ dll and the Java Native Interface (JNI) or the application code.

Debugging the ECCO C++ dll and JNI

The ECCO C++ dll uses the Java Native Interface (JNI) to access the ECCO Java classes. The dll creates a Java Virtual Machine (JVM) the first time an ECCO class is created. If the dll cannot create the JVM or, as classes are created, if any of the four basic ECCO classes cannot be found, the dll will print a message to stdout and exit. If the dll exits, check to see that jvm.dll is in the path and that ECCO_CLASSPATH is set correctly with the location of EccoAPI.jar and the other required jars.

After creating the JVM and finding the ECCO classes, the dll creates a java object for each C++ class. If it cannot create the java object, then the object will be NULL within the class. As methods are called, each class will try again to create the java object. The ECCOClient request methods will return ECCO_JNI_EXCEPTION, if the ECCOClient java object cannot be created. The ECCOClient method *IsClientObjectAlive* returns FALSE if the object is NULL.

In addition, the ECCOClient cannot be created if the ECCO.properties file is not found. The full pathname of the file should either be specified when the ECCOClient object is created, or the default ECCO.properties file should be found in the classpath specified in the environment variable ECCO_CLASSPATH.

The dll clears all JNI exceptions which would otherwise interfere with later method calls. The printing of JNI exceptions to stdout is turned off by default. To turn on the printing of JNI exceptions for any ECCO class, call the ECCOClient method *SetPrintJNIExceptions(TRUE).*

Setting up logging for development

The ECCO API uses the SimpleSocketServer task to log to a user-specified file, which contains a single day's messages. The file is truncated daily at the first message received after midnight, and the previous day's messages are saved in a file with a date extension. API logging is controlled by an ECCO properties file, whose options may be modified by method calls within the application. For additional information about logging from the application, see "Logging during development" on page 25.

Setting up the SimpleSocketServer

The SimpleSocketServer task must be running in order to log to a file. Use StartSimpleSocketServer.bat to start the task. The delivered SimpleSocketServerECCO.properties writes the log files to c:\edgil\data\ecco. Edit this file to specify a different directory structure or a different java classpath. For information on editing the properties file, see "Appendix D: Editing Properties Files" on page 199.

ECCO.properties

The ECCO.properties file is delivered with logging set to report only fatal errors to the console; this is the default for normal operation. To set up logging for development, you can either edit the default ECCO.properties file or create another properties file that specifies logging by uncommenting the logging sections for the output you want.You must also edit the log4j.appender.SocketApp.RemoteHost line to specify either the name or the IP address of the local machine. Here the machine name is "raven":

```
log4j.appender.SocketApp.RemoteHost=raven
```

For information on editing the properties file, see "Appendix D: Editing Properties Files" on page 199. Note that if you use the default ECCO.properties file for development, you must edit it after the debugging process is finished to prevent API logging in a production environment.

Logging from ECCO classes

The ECCO classes may be used from any thread regardless of where they are created. Therefore each instantiation of an ECCO class given a session id to use for logging. When an ECCO class is created and logging is desired, you must specify the ECCO properties file and a common session id.

To use ECCO logging, the application should create its own ECCOLog object after creating the ECCOClient. This example uses the session id “TestECCO” to log the informational message “starting my app”:

```
#include "ECCOClient.h"
#include "ECCOLog.h"

ECCOClient *pxClient = new ECCOClient("c:\\edgil\\java\\
   ECCO.properties","TestECCO");
pxClient->CertifyECCO(argv[0], argv[1], argv[2]);
...

ECCOLog *pxLogMyApp = new ECCOLog("myApp", "c:\\edgil\\java\\
   ECCO.properties","TestECCO");
pxLogMyApp->SetPriority(ECCOLOG_INFO);
pxLogMyApp->Info("starting up my app");
```

ECCO C++ Class Reference

This section contains descriptions of the ECCO C++ multi-threaded class library which provides a C++ interface for communication with EdgCapture.

There are four ECCO classes provided for use by the C++ programmer: ECCOClient, MonetaryTransactionData, TransactionSelection and ECCOLog. Common constants defining ECCO values are available to the programmer in ECCO.h.

ECCOClient

Declaration

```
ECCO_API ECCOClient : public ECCO
```

Include

```
ECCOClient.h
```

Description

An ECCOClient object encapsulates a connection to the EdgCapture server. Through the ECCOClient object, the user logs onto the EdgCapture server, sends requests, and receives replies.

Constructors

ECCOClient

```
ECCOClient(const char *pztEccoPropertiesFile = NULL,
               const char * pztSessionLoggingID = NULL)
```

Construct an ECCO Client that does not log and uses the default properties file by passing neither pztEccoPropertiesFile or pztSessionLoggingID. The default ECCO properties file is ECCO.properties.

Construct an ECCO Client that does not log and uses a user-specified properties file by passing only pztEccoPropertiesFile.

Construct an ECCO Client that logs and uses a user-specified properties file by passing both pztEccoPropertiesFile and pztSessionLoggingID.

Parameters:

`pztEccoPropertiesFile` - The path and file name for the ECCO properties file; may be either ECCO.properties or a user-specified properties file.

`pztSessionLoggingID` - A unique id to tie together log messages from a single application or thread.

Methods

ECCOClient::CertifyECCO

```
int CertifyECCO(char *pztKeystorePassword,
               char *pztCertPassword
               char *pztCertAlias
               char *pztSessionLoggingID = NULL)
```

Certifies ECCO for a secure connection. This is a static call that can made once per application before logging on to the ECCO Server. The char pointers are zeroed out after use.

Parameters:

`pztKeystorePassword` - Password for the keystore.

`pztCertPassword` - Password for the local certificate.

`pztCertAlias` - Alias for the local certificate.

`pztSessionLoggingID` - A unique id to tie together log messages from a single application or thread.

Returns:

An ECCOStatusCode.

ECCOClient::GetStatusMessage

```
char *GetStatusMessage(char *pztStatusMessage,
               int iBufferSize)
```

Gets the status message from the last request sent to EdgCapture.

Parameters:

`pztStatusMessage` - Buffer to hold the status message from the last request sent to the server.

`iBufferSize` - Size of passed buffer.

Returns:

`pztStatusMessage`

ECCOClient::IsClientObjectAlive

```
bool IsClientObjectAlive()
```

Checks to see that the ECCOClient object has been created in the JVM.

Parameters:

None

Returns:

TRUE if the ECCOClient object exists. FALSE if the object is NULL.

ECCOClient::Logon

```
int Logon(const char *pztLogonID,
               const char *pztPassword,
               bool bAutomatic = FALSE)
```

Logs on to the ECCO Server with a Client specified setting of manual or automatic for monetary transactions.

Parameters:

`pztLogonID` - The client's user id for identification with the ECCO Server

`pztPassword` - The client's password for identification with the ECCO Server

`bAutomatic` - If TRUE, specify automatic for monetary transactions. IF FALSE, specify manual for monetary transactions.

Returns:

An ECCOStatusCode.

ECCOClient::Logon

```
int Logon(const char *pztLogonID,
               const char *pztPassword,
               *pztProcessSource,
               bool bAutomatic = FALSE)
```

Logs on to the ECCO Server with a user-specified setting of Manual or Automatic for Manual Transactions and a client specified process source.

Parameters:

`pztLogonID` - The client's user id for identification with the ECCO Server

`pztPassword` - The client's password for identification with the ECCO Server

`pztProcessSource` - The client's process identifier for consolidating multiple users within an asynchronous application.

`bAutomatic` - TRUE specifies Automatic for Monetary Transactions; FALSE specifies Manual for Monetary Transactions.

Returns:

An ECCOStatusCode.

ECCOClient::Logoff

```
void Logoff(void)
```

Closes the Connection to the ECCO Server

ECCOClient::RequestAuthorization

`int RequestAuthorization(MonetaryTransactionData &theData)`

Submits a transaction for authorization. Sends a request to EdgCapture to obtain authorization for the sale represented by the passed MonetaryTransactionData.

Parameters:

`theData` - MonetaryTransactionData object with the minimum of the following required information provided: merchant ID, OEP ID, Transaction ID, amount, ECI, payment fields for credit card or check or direct debit.

Returns:

An ECCOStatusCode. If ECCO_SUCCESS, the transaction will be authorized in the EdgCapture database. The MonetaryTransactionData object will contain the authorization code and date, the reference number and other EdgCapture information.

ECCOClient::RequestCredit

`int RequestCredit(MonetaryTransactionData &theData)`

Submits a credit for capture. Sends a request to EdgCapture to process the credit represented by the passed MonetaryTransactionData. Successful credits are placed in the marked for capture queue.

Parameters:

`theData` - MonetaryTransactionData object with the minimum of the following required information provided: merchantId, oepId, transactionId, amount, account number, expiration month, expiration year, ECI.

Returns:

An ECCOStatusCode. If ECCO_SUCCESS, the transaction will be marked for capture in the EdgCapture database. The MonetaryTransactionData object will contain all the EdgCapture information.

ECCOClient::RequestDirectory

`int RequestDirectory(TransactionSelection &theDirectory)`

Obtains transaction data from EdgCapture. Sends a request to EdgCapture for data from a single specified transaction or for a directory of transactions.

Parameters:

`theDirectory` - TransactionSelection object with any desired selection limitation set.

Returns:

An ECCOStatusCode. If ECCO_SUCCESS, the TransactionSelection object will contain the returned list of MonetaryTransactionData objects. Use the TransactionSelection's methods GetNumberRowsReturned and GetNextTransaction for access to the MonetaryTransactionData objects.

ECCOClient::RequestMarkForCapture

`int **RequestMarkForCapture**(MonetaryTransactionData &theData)`

Marks an authorized sale for capture. Sends a request to EdgCapture to mark for capture the authorized sale represented by the passed MonetaryTransactionData.

Parameters:

`theData` - MonetaryTransactionData object with the minimum of the following required information provided: merchantId, oepId, transactionId.

Returns:

An ECCOStatusCode. If ECCO_SUCCESS, the transaction will be marked for capture in the EdgCapture database. The MonetaryTransactionData object will contain all the EdgCapture information.

ECCOClient::RequestRecordOnly

`int **RequestRecordOnly**(MonetaryTransactionData &theData)`

Submits transaction data for entry into the database without sending it to the payment network. Sends a request to EdgCapture to enter the transaction represented by the passed MonetaryTransactionData for recording purposes.

Parameters:

`theData` - MonetaryTransactionData object with the minimum of the following required information provided: merchantId, oepId, transactionId, amount, accountNumber. Note that expiration date information is not required.

Returns:

An ECCOStatusCode. If ECCO_SUCCESS, the transaction will be entered in the EdgCapture database.

ECCOClient::RequestUpdateUserData

`int RequestUpdateUserData(MonetaryTransactionData &theData)`

Updates all UserData fields and the Notes field for the specified transaction. theData is usually an object previously used in another transaction or returned from a call to RequestDirectory.

Parameters:

theData – MonetaryTransactionData object with the minimum of the following required information provided:merchantID, oepID, transactionID, UserData1-UserData10,Notes.

Returns:

An ECCOStatusCode. If ECCO_SUCCESS, the user data for the transaction has been updated.

ECCOClient::RequestVoid

`int **RequestVoid**(MonetaryTransactionData &theData)`

Voids a transaction in the EdgCapture database that is not yet captured. Sends a request to EdgCapture to void the transaction represented by the passed MonetaryTransactionData. Voids are only possible before the transaction has been captured and moved from the MarkedForCapture queue to the Captured queue. Because voids require all available data for the transaction to be voided, you must first retrieve the MonetaryTransactionData object from EdgCapture using the TransactionSelection RequestDirectory method.

Parameters:

`theData` - MonetaryTransactionData object with all available data for the transaction to be voided.

Returns:

An ECCOStatusCode. If ECCO_SUCCESS, the transaction will be moved from the Authorized or MarkedForCapture queue to the Voided queue. The MonetaryTransactionData object will contain all the EdgCapture information.

ECCOClient::SetECCOPriority

`void SetECCOPriority(int iPriority)`

Resets the priority for logging by all of ECCO.

Parameters:

`iPriority` - One of the three logging priority levels: ECCOLOG_INFO, ECCOLOG_WARN, or ECCOLOG_ERROR.

ECCOClient::SetPrintJNIException

`void SetPrintJNIException(bool bPrintJNIExceptions)`

Sets JNI exceptions to print or not.

Parameters:

`bPrintJNIExceptions` - TRUE to print JNI exceptions; FALSE to set printing off.

ECCOLog

Declaration

```
ECCO_API ECCOLog : public ECCO
```

Include

```
ECCOLog.h
```

Description

ECCOLog provides optional logging for the ECCO API and the users of the ECCO API. The user of ECCOLog may change the Priority of logging at any time to turn logging on or off.

Constructors

ECCOLog

```
ECCOLog(const char *pztIdentity,
               const char *pztEccoPropertiesFile = NULL,
               const char *pztSessionLoggingID = NULL);
```

Sets up logging for debugging using the Identity string in the log record.

Parameters:

`pztIdentity` - Name to be included in the debugging log record.

`pztEccoPropertiesFile` - Full pathname of ECCO properties file.

`pztSessionLoggingID` - String to tie together records from this thread.

Methods

ECCOLog::Error

```
void Error(const char *pztLogMessage)
```

Writes the passed message to the debugging log record if logging is active at this priority level.

Parameters:

`pztLogMessage` - Message to be written to the debugging log record.

ECCOLog::Warn

```
void Warn(const char *pztLogMessage)
```

Writes the passed message to the debugging log record if logging is active at this priority level.

Parameters:

`pztLogMessage` - Message to be written to the debugging log record.

ECCOLog::Info

`void **Info**(const char *pztLogMessage)`

Writes the passed message to the debugging log record if logging is active at this priority level.

Parameters:

`pztLogMessage` - Message to be written to the debugging log record.

ECCOLog::SetPriority

`void **SetPriority**(int iPriority)`

Set debugging logging priority to the level passed in iPriority.

Parameters:

`iPriority` - One of the three logging priority levels: ECCOLOG_INFO, ECCOLOG_WARN, or ECCOLOG_ERROR.

MonetaryTransactionData

Declaration

```
ECCO_API MonetaryTransactionData : public ECCO
```

Include

```
MonetaryTransactionData.h
```

Description

A MonetaryTransactionData object contains the data for a monetary transaction sent to or returned from EdgCapture. The available fields correspond to the transaction data stored in the EdgCapture database. For complete information on transaction data fields in EdgCapture, see “” on page 189.

Constructors

MonetaryTransactionData

```
MonetaryTransactionData(const char *pztECCOPropertiesFile = NULL,
                        const char *pztSessionLoggingID = NULL)
```

Construct a MonetaryTransactionData object that does not log and uses the default properties file by passing neither pztEccoPropertiesFile or pztSessionLoggingID. The default ECCO properties file is ECCO.properties.

Construct MonetaryTransactionData object that does not log and uses a user-specified properties file by passing only pztEccoPropertiesFile.

Construct MonetaryTransactionData object that logs and uses a user-specified properties file by passing both pztEccoPropertiesFile and pztSessionLoggingID.

Parameters:

`pztEccoPropertiesFile` - The path and file name for the ECCO properties file; may be either ECCO.properties or a user-specified properties file.

`pztSessionLoggingID` - A unique id to tie together log messages from a single application or thread.

Methods

MonetaryTransactionData::ClearAllFields

```
void ClearAllFields(void)
```

Clears all fields in the current MonetaryTransactionData object in order to use it for a new transaction.

MonetaryTransactionData::operator =

`MonetaryTransactionData = copySourceMonetaryTransactionData`

Copies all values from the source MonetaryTransactionData to this MonetaryTransactionData

Parameters:

`copySourceMonetaryTransactionData` - Source for copy

MonetaryTransactionData::GetABANumber

`char *`**`GetABANumber`**`(char *pztReceiveBuffer)`

Returns the ABA Number for this transaction in the receive buffer or a null string if an ABA Number is not part of the transaction's Account.

Parameters:

`pztReceiveBuffer` - Buffer to hold the ABA Number. The minimum size of the buffer is 10.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetAccountNumber

`char *`**`GetAccountNumber`**`(char *pztReceiveBuffer)`

Returns the Account Number for this transaction in the receive buffer or a null string if the account number is not available.

Parameters:

`pztReceiveBuffer` - Buffer to hold the Account Number. The minimum size of the buffer is 49.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetAddressLine1

`char *`**`GetAddressLine1`**`(char *pztReceiveBuffer)`

Returns address line 1 for this transaction in the receive buffer or a null string if not available. Maximum size is 100.

Parameters:

`pztReceiveBuffer` - Buffer to hold address line 1. The minimum size of the buffer is 101.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetAddressLine2

`char *`**`GetAddressLine2`**`(char *pztReceiveBuffer)`

Returns address line 2 for this transaction in the receive buffer or null string if not available. Maximum size is 100.

Parameters:

`pztReceiveBuffer` **-** Buffer to hold address line 2. The minimum size of the buffer is 101.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetAddressVerification

`char *`**`GetAddressVerification`**`(char *pztReceiveBuffer)`

Returns the address verification results for a transaction sent out to a credit network for authorization. A blank will be returned when address verification results are not available.

Parameters:

`pztReceiveBuffer` **-** Buffer to hold the address verification results. The minimum size of the buffer is 2.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetAmount

`char *`**`GetAmount`**`(char *pztReceiveBuffer)`

Returns the amount of the transaction in EdgCapture in the receive buffer. The amount is always positive. Maximum value is 12.2

Parameters:

`pztReceiveBuffer` **-** Buffer to hold amount. The minimum size of the buffer is 16.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetAuthorizationCode

`char *`**`GetAuthorizationCode`**`(char *pztReceiveBuffer)`

Return the authorization code of this transaction in the receive buffer or a null string if not available.

Parameters:

`pztReceiveBuffer` **-** Buffer to hold authorization code. The minimum size of the buffer is 11.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetAuthorizationDate

`char *`**`GetAuthorizationDate`**`(char *pztReceiveBuffer)`

Return the date this transaction was authorized by the credit network in the receive buffer or a null string if not available. Format is YYYY-MM-DD.

Parameters:

`pztReceiveBuffer` - Buffer to hold authorization date. The minimum size of the buffer is 11.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetAuthorizationString

`char *`**`GetAuthorizationString`**`(char *pztReceiveBuffer)`

Return the authorization code and date of authorization of this transaction in the receive buffer or a null string if not available The date of authorization is year-month-day in the form YYYY-MM-DD.

Parameters:

`pztReceiveBuffer` - Buffer to hold authorization string. The minimum size of the buffer is 21.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetCaptureDate

`char *`**`GetCaptureDate`**`(char *pztReceiveBuffer)`

Returns the capture date of the transaction as YYYY-MM-DD in the receive buffer or a null string if not available.

Parameters:

`pztReceiveBuffer` - Buffer to hold the capture date. The minimum size of the buffer is 11.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetCity

`char *`**`GetCity`**`(char *pztReceiveBuffer)`

Returns the city for this transaction in the receive buffer or a null string if not available.

Parameters:

`pztReceiveBuffer` - Buffer to hold city. The minimum size of the buffer is 51.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetECI

`char *GetECI(char *pztReceiveBuffer)`

Returns the electronic commerce indicator for this transaction in the receive buffer.

Parameters:

`pztReceiveBuffer` - Buffer to hold electronic commerce indicator. The minimum size of the buffer is 1.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetExpirationDate

`char *GetExpirationDate(char *pztReceiveBuffer)`

Returns the expiration date as MMYY in the receive buffer or a null string if not available.

Parameters:

`pztReceiveBuffer` - Buffer to hold address line 1. The minimum size of the buffer is 5.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetExpirationMonth

`char *GetExpirationMonth(char *pztReceiveBuffer)`

Returns the two-digit expiration month (0-12) for this transaction in the receive buffer or a null string if not available.

Parameters:

`pztReceiveBuffer` - Buffer to hold expiration month. The minimum size of the buffer is 3.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetExpirationYear

`void *GetExpirationYear(char *pztReceiveBuffer)`

Returns the two-digit expiration year for this transaction in the receive buffer or a null string if not available.

Parameters:

`pztReceiveBuffer` - Buffer to hold expiration year. The minimum size of the buffer is 3.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetFirstName

`char *GetFirstName(char *pztReceiveBuffer)`

Returns the first name in the receive buffer or a null string if not available.

Parameters:

`pztReceiveBuffer` - Buffer to hold first name. The minimum size of the buffer is 61.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetFraudSecurityResponse

`int GetFraudSecurityResponse()`

Returns the fraud security response code. See “FraudSecurityResponse” on page 196 for descriptions of codes.

MonetaryTransactionData::GetMerchantId

`char *GetMerchantId(char *pztReceiveBuffer)`

Returns the MerchantId associated with this transaction. The MerchantId is a fixed String identifying the Merchant to EdgCapture. The Merchant represents the bank account for the transaction funds. Maximum size is 2.

Parameters:

`pztReceiveBuffer` - Buffer to hold merchant. The minimum size of the buffer is 3.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetMiddleInitial

`char *GetMiddleInitial(char *pztReceiveBuffer)`

Returns the middle initial of the name in the receive buffer or a null string if not available Maximum size is 2.

Parameters:

`pztReceiveBuffer` - Buffer to hold middle initial. The minimum size of the buffer is 3

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetName

`char *GetName(char *pztReceiveBuffer)`

Returns the name for this transaction in the receive buffer or a null string if not available. Name is FirstName MiddleInitial LastName.

Parameters:

`pztReceiveBuffer` **-** Buffer to hold name. The minimum size of the buffer is 187.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetNotes

`char *GetNotes(char *pztReceiveBuffer)`

Returns Notes, 80 characters of client specified information or a null string if not available.

Parameters:

`pztReceiveBuffer` **-** Buffer to hold notes. The minimum size of the buffer is 81.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetOEPId

`char *GetOEPId(char *pztReceiveBuffer)`

Returns the OEP Id associated with this transaction. The OEP Id is a fixed string identifying a subaccount for a Merchant. Maximum size is 2.

Parameters:

`pztReceiveBuffer` **-** Buffer to hold OEP Id. The minimum size of the buffer is 3.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetPaymentDesignator

`char *GetPaymentDesignator(char *pztReceiveBuffer)`

Returns the payment designator for this transaction or a null string if a payment designator is not part of the transaction's Account. Values are the ECCO constants CHECK_DESIGNATOR_CORPORATE or CHECK_DESIGNATOR_CONSUMER.

Parameters:

`pztReceiveBuffer` **-** Buffer to hold payment designator. The minimum size of the buffer is 20.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetPaymentType

`char *`**`GetPaymentType`**`(char *pztReceiveBuffer)`

Returns the payment type for this transaction or a null string if not available. Value is one of the ECCO constants: PAY_AMERICAN_EXPRESS,PAY_CARTE_BLANCHE, PAY_CHECK_DRAFT, PAY_DINERS_CLUB, PAY_DIRECT_DEBIT, PAY_DISCOVER, PAY_MASTER_CARD, PAY_PAPER_CHECK, PAY_VISA

Parameters:

`pztReceiveBuffer` - Buffer to hold payment type. The minimum size of the buffer is 31.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetQueueDateTime

`char *`**`GetQueueDateTime`**`(char *pztReceiveBuffer)`

Returns the datetime of entry for the transaction's queue in EdgCapture in the receive buffer or a null string if not available. The date of entry is year-month-day hour-minute-second in the form YYYY-MM-DD HH:MM:SS. Transactions move from queue to queue as they are processed on EdgCapture.

Parameters:

`pztReceiveBuffer` - Buffer to hold queue date time. The minimum size of the buffer is 20.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetQueueName

`char *`**`GetQueueName`**`(char *pztReceiveBuffer)`

Returns the name of the transaction's queue in EdgCapture in the receive buffer or a null string if not available. Transactions move from queue to queue as they are processed on EdgCapture. ECCO constants for the set of queues: QUEUE_AUTHORIZED, QUEUE_CAPTURED, QUEUE_DECLINED, QUEUE_ERROR, QUEUE_HELD, QUEUE_INPUT, QUEUE_MARKED_FOR_CAPTURE, QUEUE_RECORDED_ONLY, QUEUE_UNKNOWN_DISPOSITION, QUEUE_VOIDED.

Parameters:

`pztReceiveBuffer` - Buffer to hold queue name. The minimum size of the buffer is 31.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetQueueString

`char *GetQueueString(char *pztReceiveBuffer)`

Returns the name and datetime of entry for the transaction's queue in EdgCapture in the receive buffer or a null string if not available. The datetime of entry is year-month-day hour-minute-second in the form YYYY-MM-DD HH-MM-SS. Transactions move from queue to queue as they are processed on EdgCapture. ECCO constants for the set of queues: QUEUE_AUTHORIZED, QUEUE_CAPTURED, QUEUE_DECLINED, QUEUE_ERROR, QUEUE_HELD, QUEUE_INPUT, QUEUE_MARKED_FOR_CAPTURE, QUEUE_RECORDED_ONLY, QUEUE_UNKNOWN_DISPOSITION, QUEUE_VOIDED.

Parameters:

`pztReceiveBuffer` **-** Buffer to hold queue string. The minimum size of the buffer is 50.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetReferenceNumber

`char *GetReferenceNumber(char *pztReceiveBuffer)`

Returns the reference number generated by the credit network or EdgCapture at authorization time or a null string if not available.

Parameters:

`pztReceiveBuffer` **-** Buffer to hold reference number. The minimum size of the buffer is 11.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetSalesTax

`char *GetSalesTax(char *pztReceiveBuffer)`

Returns the sales tax amount in the receive buffer. Format is nnnn.nn (maximum size 12,2). Note that Amount contains the total transaction amount including sales tax.

Parameters:

`pztReceiveBuffer` **-** Buffer to hold sales tax amount. The minimum size of the buffer is 15.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetSource

`char *GetSource(char *pztReceiveBuffer)`

Returns the Process source in the receive buffer or a null string if not available.

Parameters:

`pztReceiveBuffer` **-** Buffer to hold process source. The minimum size of the buffer is 41.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetStateOrProvince

`char *`**`GetStateOrProvince`**`(char *pztReceiveBuffer)`

Returns the customer address state or province for this transaction in the receive buffer or a null string if not available.

Parameters:

`pztReceiveBuffer` **-** Buffer to hold state or province. The minimum size of the buffer is 3.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetTelephoneNumber

`char *`**`GetTelephoneNumber`**`(char *pztReceiveBuffer)`

Returns the telephone number for this transaction in the receive buffer or a null string if not available.

Parameters:

`pztReceiveBuffer` **-** Buffer to hold phone number. The minimum size of the buffer is 17.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetTransactionId

`char *`**`GetTransactionId`**`(char *pztReceiveBuffer)`

Returns the Transaction Id that identifies the transaction uniquely within Merchant and OEP.

Parameters:

`pztReceiveBuffer` **-** Buffer to hold the transaction Id. The minimum size of the buffer is 31.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetTransactionStatusCode

`int `**`GetTransactionStatusCode`**`()`

Returns message code for the status of the transaction in EdgCapture. Returns -1 if code not available. See "Appendix B: ECCO Status Codes" on page 189 for a complete listing of status codes.

MonetaryTransactionData::GetTransactionStatusMessage

`char *GetTransactionStatusMessage(char *pztReceiveBuffer)`

Returns the message text for the status of the transaction in EdgCapture in the receive buffer. Maximum size is determined by the longest message in ECCOStatusMessages.properties.

Parameters:

`pztReceiveBuffer` **-** Buffer to hold status message. The minimum size of the buffer is determined by the longest message in ECCOStatusMessages.properties.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetTransactionStatusString

`char *GetTransactionStatusString(char *pztReceiveBuffer)`

Returns the message code and text for the status of the transaction in EdgCapture. The format is the message code, a blank, the message string. Maximum size is determined by the longest message in ECCOStatusMessages.properties plus 5.

Parameters:

`pztReceiveBuffer` **-** Buffer to hold transaction status string. The minimum size of the buffer is determined by the longest message in ECCOStatusMessages.properties plus 5.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetTransactionType

`char *GetTransactionType(char *pztReceiveBuffer)`

Returns the transaction type from ECCOClient as one of the ECCO constants: TRAN_AUTHORIZE, TRAN_CAPTURE_HELD_TRANSACTION, TRAN_CREDIT, TRAN_DECLINE_HELD_TRANSACTION, TRAN_MARKED_FOR_CAPTURE, TRAN_RECORD_ONLY, TRAN_VOID.

Parameters:

`pztReceiveBuffer` **-** Buffer to hold transaction type. The minimum size of the buffer is 31.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetUserData1

`char *GetUserData1(char *pztReceiveBuffer)`

Returns UserData1 in the receive buffer or a null string if not available. UserData1 is 40 characters of client specified information.

Parameters:

`pztReceiveBuffer` **-** Buffer to hold user data 1. The minimum size of the buffer is 41.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetUserData10

`char *`**`GetUserData10`**`(char *pztReceiveBuffer)`

Returns UserData10 in the receive buffer or a null string if not available. UserData10 is 40 characters of client specified information.

Parameters:

`pztReceiveBuffer` - Buffer to hold user data 10. The minimum size of the buffer is 41

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetUserData2

`char *`**`GetUserData2`**`(char *pztReceiveBuffer)`

Returns UserData2 in the receive buffer or a null string if not available. UserData2 is 40 characters of client specified information.

Parameters:

`pztReceiveBuffer` - Buffer to hold user data 2. The minimum size of the buffer is 41

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetUserData3

`char *`**`GetUserData3`**`(char *pztReceiveBuffer)`

Returns UserData3 in the receive buffer or a null string if not available. UserData3 is 40 characters of client specified information.

Parameters:

`pztReceiveBuffer` - Buffer to hold user data 3. The minimum size of the buffer is 41

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetUserData4

`char *`**`GetUserData4`**`(char *pztReceiveBuffer)`

Returns UserData4 in the receive buffer or a null string if not available. UserData4 is 40 characters of client specified information.

Parameters:

`pztReceiveBuffer` - Buffer to hold user data 4. The minimum size of the buffer is 41

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetUserData5

`char *GetUserData5(char *pztReceiveBuffer)`

Returns UserData5 in the receive buffer or a null string if not available. UserData5 is 40 characters of client specified information.

Parameters:

`pztReceiveBuffer` - Buffer to hold user data 5. The minimum size of the buffer is 41

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetUserData6

`char *GetUserData6(char *pztReceiveBuffer)`

Returns UserData6 in the receive buffer or a null string if not available. UserData6 is 40 characters of client specified information.

Parameters:

`pztReceiveBuffer` - Buffer to hold user data 6. The minimum size of the buffer is 41

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetUserData7

`char *GetUserData7(char *pztReceiveBuffer)`

Returns UserData7 in the receive buffer or a null string if not available. UserData7 is 40 characters of client specified information.

Parameters:

`pztReceiveBuffer` - Buffer to hold user data 7. The minimum size of the buffer is 41

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetUserData8

`char *GetUserData8(char *pztReceiveBuffer)`

Returns UserData8 in the receive buffer or a null string if not available. UserData8 is 40 characters of client specified information.

Parameters:

`pztReceiveBuffer` - Buffer to hold user data 8. The minimum size of the buffer is 41

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetUserData9

`char *GetUserData9(char *pztReceiveBuffer)`

Returns UserData9 in the receive buffer or a null string if not available. UserData9 is 40 characters of client specified information.

Parameters:

`pztReceiveBuffer` - Buffer to hold user data 9. The minimum size of the buffer is 41

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::GetZipCode

`char *GetZipCode(char *pztReceiveBuffer)`

Returns zip code for this transaction in the receive buffer or a null string if not available.

Parameters:

`pztReceiveBuffer` - Buffer to hold zip code. The minimum size of the buffer is 12.

Returns:

`pztReceiveBuffer`

MonetaryTransactionData::SetABANumber

`void SetABANumber(const char *pztABANumber)`

Sets the ABA Number as part of the REQUIRED payment information for an authorize request. When ABA Number is set individually, the other related fields must also be set: AccountNumber, PaymentDesignator, PaymentType. ABA Number is used with Payment Types PAY_DIRECT_DEBIT and PAY_CHECK_DRAFT. Maximum size is 9.

Parameters:

`pztABANumber` - The bank routing number for a checking account.

MonetaryTransactionData::SetAccountNumber

`void SetAccountNumber(const char *pztAccountNumber)`

Sets the Account Number as part of the REQUIRED payment information for an authorize or credit request. When payment type is set individually, the other related fields must also be set. When Payment type is PAY_CREDIT_CARD or a specific card type (ECCO constants: PAY_AMERICAN_EXPRESS, PAY_CARTE_BLANCHE, PAY_DINERS_CLUB, PAY_DISCOVER, PAY_MASTER_CARD, PAY_VISA), Payment Type and ExpirationDate must also be set. When Payment Type is PAY_DIRECT_DEBIT or PAY_CHECK_DRAFT, Payment Type, ABANumber and PaymentDesignator must also be set. Maximum size is 48.

Parameters:

`pztAccountNumber` - For credit cards, the credit card number, for direct debit and check draft, the account number.

MonetaryTransactionData::SetAddress

```
void SetAddress(const char *pztCity,
                const char *pztStateOrProvince,
                const char *pztZipCode)
```

Sets the OPTIONAL address values for city, state or province and zip code.

Parameters:

`pztCity` - The customer city. Maximum size is 50.

`pztStateOrProvince` - The state or province. Maximum size is 2.

`pztZipCode` - The zipCode or postal code. Maximum size is 12. Only alphas and numerics permitted.

MonetaryTransactionData::SetAddress

```
void SetAddress(const char *pztAddressLine1,
                const char *pztCity,
                const char *pztStateOrProvince,
                const char *pztZipCode)
```

Sets the OPTIONAL address values for address line one, city, state or province and zip code.

Parameters:

`pztAddressLine1` - The first line of the customer's address. Maximum size is 100.

`pztCity` - The customer city. Maximum size is 50.

`pztStateOrProvince` - The state or province. Maximum size is 2.

`pztZipCode` - The zipCode or postal code. Maximum size is 12. Only alphas and numerics permitted.

MonetaryTransactionData::SetAddressLine1

```
void SetAddressLine1(const char *pztaddressLine1)
```

Sets the value of OPTIONAL address line one. Maximum size is 100.

Parameters:

`pztAdressLine1` - The first line of the customer's address. Maximum size is 100.

MonetaryTransactionData::SetAddressLine2

```
void SetAddressLine2(const char *pztAddressLine2)
```

Sets the value of OPTIONAL address line two. Maximum size is 100.

Parameters:

`pztAddressLine2` - The second line of the customer's address. Maximum size is 100.

MonetaryTransactionData::SetAmount

```
void SetAmount(const char *pztAmount)
```

Sets the REQUIRED amount of the transaction for an authorize or credit request. The amount must be entered as a decimal value: nnn.nn. The decimal point and the two digits after the decimal point are required. The amount is always positive whether the transaction is a Sale or a Credit. Maximum size is 12,2.

Parameters:

`pztAmount`- Amount of the transaction in the format: nnnnn.nn.

MonetaryTransactionData::SetCheckDraft

```
void SetCheckDraft(const char *pztAccountNumber,
               const char *pztABANumber,
               const char *pztPaymentDesignator)
```

Sets all REQUIRED check draft payment information for an authorize or credit request.

Parameters:

`pztAccountNumber` - The check number for the check draft. Maximum size is 48.

`pztABANumber` - The bank routing number for the check draft. Maximum size is 9.

`pztPaymentDesignator` - The checking account type which must be one of the ECCO constants CHECK_DESIGNATOR_CORPORATE or CHECK_DESIGNATOR_CONSUMER

MonetaryTransactionData::SetCity

```
void SetCity(const char *pztCity)
```

Sets the value of OPTIONAL city. Maximum size is 50.

Parameters:

`pztCity` - The customer city. Maximum size is 50.

MonetaryTransactionData::SetCreditCard

```
void SetCreditCard(const char *pztAccountNumber,
               const char *pztExpirationMonth,
               const char *pztExpirationYear)
```

Sets all REQUIRED credit card payment information for an authorize or credit request.

Parameters:

`pztAcountNumber` - The account number for the credit card. Maximum size is 48.

`pztExpirationMonth` - The expiration month as two digits: nn (01 to 12)

`pztExpirationYear` - The expiration year as two digits: nn

MonetaryTransactionData::SetCreditCard

```
void SetCreditCard(const char *pztAccountNumber,
               const char *pztExpirationMonth,
               const char *pztExpirationYear,
               const char *pztPaymentType)
```

Sets all REQUIRED credit card payment information for an authorize or credit request. Also sets OPTIONAL payment type.

Parameters:

`pztAccountNumber` - The account number for the credit card. Maximum size is 48.

`pztExpirationMonth` - The expiration month as two digits: nn (01 to 12)

`pztExpirationYear` - The expiration year as two digits: nn

`pztPaymentType` - User selected payment type for cross validation with account number. ECCO constant payment types: PAY_AMERICAN_EXPRESS, PAY_CARTE_BLANCHE PAY_DINERS_CLUB, PAY_DISCOVER, PAY_MASTER_CARD, PAY_VISA

MonetaryTransactionData::SetDirectDebit

```
void SetDirectDebit(const char *pztaccountNumber,
               const char *pztABANumber,
               const char *pztPaymentDesignator)
```

Sets all REQUIRED direct debit payment information for an authorize or credit request.

Parameters:

`pztAccountNumber` - The account number for direct debit. Maximum size is 48.

`pztABANumber` - The bank routing number for direct debit. Maximum size is 9.

`pztPaymentDesignator` - The checking account type which must be one of the ECCO constants CHECK_DESIGNATOR_CORPORATE or CHECK_DESIGNATOR_CONSUMER

MonetaryTransactionData::SetECI

```
void SetECI(const char *pztECI)
```

Sets the electronic commerce indicator REQUIRED for authorize and credit requests.

Parameters:

`pztECI` - The one-character ECI, which must be one of the following values. Note that 6 is the most likely value for web-based ECCO applications.

0 - Unknown
1 - Mail Order/Telephone Order (single, non-recurring transaction)
2 - Mail Order/Telephone Order (recurring periodic transaction, such as subscription)
3 - Mail Order/Telephone Order (installment)
4 - Secure Electronic Transaction (SET) protocol using cardholder certificate management
5 - SET with no cardholder certificate management, but includes merchant certificate
6 - Non-SET, channel-encrypted security such as SSL (Web site using SSL)
7 - Non-SET, non-channel-encrypted security (clear text mail)

MonetaryTransactionData::SetExpirationDate

```
void SetExpirationDate(const char *pztExpirationMonth,
               const char *pztExpirationYear)
```

Set the expiration date as part of the AUTHORIZE and CREDIT REQUIRED payment information. When the expiration date is set individually, the other related fields must also be set: AccountNumber, PaymentType. Expiration date is used with Payment Type PAY_CREDIT_CARD and the specific ECCO constant card types: PAY_AMERICAN_EXPRESS, PAY_CARTE_BLANCHE, PAY_DINERS_CLUB, PAY_DISCOVER, PAY_MASTER_CARD, PAY_VISA.

Parameters:

`pztExpirationMonth` - The expiration month as two digits: nn (01 to 12)

`pztExpirationYear` - The expiration year as two digits: nn

MonetaryTransactionData::SetExpirationMonth

```
void SetExpirationMonth(const char *pztExpirationMonth)
```

Sets the expiration month as part of the REQUIRED payment information for an authorize or credit request. When the expiration month is set individually, the other related fields must also be set: ExpirationYear, AccountNumber, PaymentType. Expiration month is used with Payment Type PAY_CREDIT_CARD and the specific ECCO constant card types: PAY_AMERICAN_EXPRESS, PAY_CARTE_BLANCHE, PAY_DINERS_CLUB, PAY_DISCOVER, PAY_MASTER_CARD, PAY_VISA.

Parameters:

`pztExpirationMonth` - The expiration month as two digits: nn (1 to 12)

MonetaryTransactionData::SetExpirationYear

```
void SetExpirationYear(const char *pztExpirationYear)
```

Sets the expiration year as part of the REQUIRED payment information for an authorize or credit request. When the expiration year is set individually, the other related fields must also be set: ExpirationMonth, AccountNumber, PaymentType. Expiration year is used with Payment Type PAY_CREDIT_CARD and the specific ECCO constant card types: PAY_AMERICAN_EXPRESS, PAY_CARTE_BLANCHE, PAY_DINERS_CLUB, PAY_DISCOVER, PAY_MASTER_CARD, PAY_VISA.

Parameters:

`pztExpirationYear` - The expiration year as two digits: nn

MonetaryTransactionData::SetFraudSecurityValue

```
void SetFraudSecurityValue(const char *pztFraudSecurityValue)
```

Sets the value for the OPTIONAL fraud security value, a 3- or 4-digit code separate from the account number found on Visa, Master Card, American Express, and Discover cards. The value provides additional security during authorization for merchants in card-not-present transactions. Note that in accordance with card issuer's regulations, this value is not stored in the database for later retrieval.

Parameters:

`pztFraudSecurityValue` - The 3- or 4-digit fraud security value.

MonetaryTransactionData::SetMerchantId

```
void SetMerchantId(const char *pztMerchantId)
```

Sets the value for the REQUIRED MerchantId. The MerchantId is a fixed string identifying the destination bank account to EdgCapture. Maximum size is 2.

Parameters:

`pztMerchantId` - A fixed string identifying the destination bank account to EdgCapture.

MonetaryTransactionData::SetName

```
void SetName(const char *pztLastName,
               const char *pztFirstName = NULL,
               const char *pztMiddleInitial = NULL)
```

Sets the value of the OPTIONAL customer name: first name, middle initial, last name.

Parameters:

`pztLastName` - The last name of the customer or the company name. Maximum size is 60.

`pztFirstName` - The first name of the customer. Maximum size is 60.

`pztMiddleInitial` - The middle initial of the customer. Maximum size is 2.

MonetaryTransactionData::SetNotes

```
void SetNotes(const char *pztNotes)
```

Sets the value of OPTIONAL notes. Notes is 80 characters of client specified information kept on EdgCapture where it may be viewed, reported on or returned to ECCO with other transaction information.

Parameters:

`pztNotes` - Up to 80 characters of transaction notes.

MonetaryTransactionData::SetOEPId

```
void SetOEPId(const char *pztOepId)
```

Sets the value for the REQUIRED OEPId. The OEPId is a fixed string identifying the client subaccount to EdgCapture. Maximum size is 2.

Parameters:

`pztOepId` - Fixed string identifying the client subaccount. Maximum size is 2.

MonetaryTransactionData::SetPaymentDesignator

```
void SetPaymentDesignator(const char *pztPaymentDesignator)
```

Sets the payment designator as part of the REQUIRED payment information for an authorize or credit request for a direct debit or checking account. When payment designator is set individually, the other related fields must also be set: AccountNumber, ABANumber, PaymentType. Payment Designator is used with Payment Types PAY_DIRECT_DEBIT and PAY_CHECK_DRAFT.

Parameters:

`pztPaymentDesignator` - The checking account type which must be one of the ECCO constants CHECK_DESIGNATOR_CORPORATE or CHECK_DESIGNATOR_CONSUMER

MonetaryTransactionData::SetPaymentType

```
void SetPaymentType(const char *pztPaymentType)
```

Sets the payment type as part of the REQUIRED payment information for an authorize or credit request. When payment type is set individually, the other related fields must also be set. When Payment type is PAY_CREDIT_CARD or a specific ECCO constant card type (PAY_AMERICAN_EXPRESS, PAY_CARTE_BLANCHE, PAY_DINERS_CLUB, PAY_DISCOVER, PAY_MASTER_CARD, PAY_VISA), AccountNumber and ExpirationDate must also be set. When Payment Type is PAY_DIRECT_DEBIT or PAY_CHECK_DRAFT, AccountNumber, ABANumber and PaymentDesignator must also be set.

Parameters:

`pztPaymentType` - The checking account type which must be one of the ECCO constants CHECK_DESIGNATOR_CORPORATE or CHECK_DESIGNATOR_CONSUMER

MonetaryTransactionData::SetReferenceNumber

```
void SetReferenceNumber(const char *pztReferenceNumber)
```

Sets the reference number as part of the REQUIRED information for a void request. MerchantId, OEPId and TransactionId must also be set.

Parameters:

`pztReferenceNumber` - The reference number associated with an authorized transaction or credit in the EdgCapture database.

MonetaryTransactionData::SetSalesTax

```
void SetSalesTax(const char *pztSalesTax)
```

Sets the sales tax amount as part of the OPTIONAL information for an authorize request. 0 may be expressed as .00 or 0.00. Note that the Amount contains the total transaction amount, including sales tax.

Parameters:

`pztSalesTax` - The sales tax amount, in the format nnnn.nn. Maximum size is 12,2.

MonetaryTransactionData::SetStateOrProvince

`void SetStateOrProvince(const char *pztStateOrProvince)`

Sets the value of OPTIONAL state or province code. Maximum size is 2.

Parameters:

`pztStateOrProvince` - Two-character state or province code.

MonetaryTransactionData::SetTelephoneNumber

`void SetTelephoneNumber(const char *pztPhoneNumber)`

Sets the values of OPTIONAL telephone number. Maximum size is 16. No formatting is required.

Parameters:

`pztPhoneNumber` - Unformatted phone number string. Maximum size is 16.

MonetaryTransactionData::SetTransactionId

`void SetTransactionId(const char *pztTransactionId)`

Sets the value of the REQUIRED Transaction Id which must be unique within the Merchant and OEP. Maximum size is 30.

Parameters:

`pztTransactionId` - string identifying transaction. Maximum size is 30.

MonetaryTransactionData::SetUserData1

`void SetUserData1(const char *pztUserData)`

Sets the value of OPTIONAL UserData1. UserData1 is 40 characters of client specified information kept on EdgCapture where it may be viewed, reported on or returned to ECCO with other transaction information.

Parameters:

`pztUserData` - Up to 40 characters of user-specified data.

MonetaryTransactionData::SetUserData10

`void SetUserData10(const char *pztUserData)`

Sets the value of OPTIONAL UserData10. UserData10 is 40 characters of client specified information kept on EdgCapture where it may be viewed, reported on or returned to ECCO with other transaction information.

Parameters:

`pztUserData` - Up to 40 characters of user-specified data.

MonetaryTransactionData::SetUserData2

`void SetUserData2(const char *pztUserData)`

Sets the value of OPTIONAL UserData2. UserData2 is 40 characters of client specified information kept on EdgCapture where it may be viewed, reported on or returned to ECCO with other transaction information.

Parameters:

`pztUserData` - Up to 40 characters of user-specified data.

MonetaryTransactionData::SetUserData3

`void SetUserData3(const char *pztUserData)`

Sets the value of OPTIONAL UserData3. UserData3 is 40 characters of client specified information kept on EdgCapture where it may be viewed, reported on or returned to ECCO with other transaction information.

Parameters:

`pztUserData` - Up to 40 characters of user-specified data.

MonetaryTransactionData::SetUserData4

`void SetUserData4(const char *pztUserData)`

Sets the value of OPTIONAL UserData4. UserData3 is 40 characters of client specified information kept on EdgCapture where it may be viewed, reported on or returned to ECCO with other transaction information.

Parameters:

`pztUserData` - Up to 40 characters of user-specified data.

MonetaryTransactionData::SetUserData5

`void SetUserData5(const char *pztUserData)`

Sets the value of OPTIONAL UserData5. UserData5 is 40 characters of client specified information kept on EdgCapture where it may be viewed, reported on or returned to ECCO with other transaction information.

Parameters:

`pztUserData` - Up to 40 characters of user-specified data.

MonetaryTransactionData::SetUserData6

`void SetUserData6(const char *pztUserData)`

Sets the value of OPTIONAL UserData6. UserData6 is 40 characters of client specified information kept on EdgCapture where it may be viewed, reported on or returned to ECCO with other transaction information.

Parameters:

`pztUserData` - Up to 40 characters of user-specified data.

MonetaryTransactionData::SetUserData7

`void SetUserData7(const char *pztUserData)`

Sets the value of OPTIONAL UserData7. UserData7 is 40 characters of client specified information kept on EdgCapture where it may be viewed, reported on or returned to ECCO with other transaction information.

Parameters:
`pztUserData` - Up to 40 characters of user-specified data.

MonetaryTransactionData::SetUserData8

`void SetUserData8(const char *pztUserData)`

Sets the value of OPTIONAL UserData8. UserData8 is 40 characters of client specified information kept on EdgCapture where it may be viewed, reported on or returned to ECCO with other transaction information.

Parameters:
`pztUserData` - Up to 40 characters of user-specified data.

MonetaryTransactionData::SetUserData9

`void SetUserData9(const char *pztUserData)`

Sets the value of OPTIONAL UserData9. UserData9 is 40 characters of client specified information kept on EdgCapture where it may be viewed, reported on or returned to ECCO with other transaction information.

Parameters:
`pztUserData` - Up to 40 characters of user-specified data.

MonetaryTransactionData::SetZipCode

`void SetZipCode(const char *pztZipCode)`

Sets the value of OPTIONAL zip code. Note, however, that zip code is required for address verification. Only alphanumeric characters are permitted. Maximum size is 12.

Parameters:
`pztZipCode` - Zip code or postal code, expressed as alphanumeric string. Maximum size is 12.

TransactionSelection

Declaration

```
ECCO_API TransactionSelection : public ECCO
```

Include

```
TransactionSelection.h
```

Description

A TransactionSelection object contains the selection criteria used to retrieve transaction information from the EdgCapture database

Constructors

TransactionSelection

```
TransactionSelection(*pztSessionLoggingID = NULL,
                     *pztPropertiesFile = NULL)
```

Construct a TransactionSelection object that does not log and uses the default properties file by passing neither pztEccoPropertiesFile or pztSessionLoggingID. The default ECCO properties file is ECCO.properties.

Construct TransactionSelection object that does not log and uses a user-specified properties file by passing only pztEccoPropertiesFile.

Construct TransactionSelection object that logs and uses a user-specified properties file by passing both pztEccoPropertiesFile and pztSessionLoggingID.

Methods

TransactionSelection::ClearAllEntries

```
void ClearAllEntries(void)
```

Clears all previously set selection criteria.

TransactionSelection::ClearAllEntriesExceptMerchant

```
void ClearAllEntriesExceptMerchant(void)
```

Clears all previously set selection criteria except the Merchant(s) and their OEPs.

TransactionSelection::SetSelectionTransactionKey

```
void SetSelectionTransactionKey(const char *pztMerchantId,
              const char *pztOepId,
              const char *pztTransactionId)
```

Sets the values for the three fields REQUIRED to select a unique transaction: Merchant, OEP, TransactionId.

Parameters:

`pztMerchantId` - Fixed string identifying the destination bank account to EdgCapture.

`pztOepId` - Fixed string identifying the client subaccount to EdgCapture.

`pztTransactionId` - String that uniquely identifies the transaction within the Merchant and the OEP.

TransactionSelection::SetSelectionTransactionKeyMerchant

```
void SetSelectionTransactionKeyMerchant(const char *pztMerchantId)
```

Sets the value of the REQUIRED MerchantId for selecting a unique transaction using the Key: Merchant, OEP, TransactionId. The merchant Id is a fixed string identifying the destination bank account to EdgCapture.

Parameters:

`pztMerchantId` - Fixed string identifying the destination bank account to EdgCapture.

TransactionSelection::SetSelectionTransactionKeyOep

```
void SetSelectionTransactionKeyOep(const char *pztOepId)
```

Sets the value of the REQUIRED OEPId for selecting a unique transaction using the Key: Merchant, OEP, Transaction Id.

Parameters:

`pztOepId` - Fixed string identifying the client subaccount to EdgCapture.

TransactionSelection::SetSelectionTransactionKeyTransactionId

```
void SetSelectionTransactionKeyTransactionId(
              const char *pztTransactionId)
```

Sets the value of the REQUIRED Transaction Id for selecting a unique transaction using the Key: Merchant, OEP, Transaction Id.

Parameters:

`pztTransactionId` - String that uniquely identifies the transaction within the Merchant and the OEP.

Methods available with purchase of the directory option

TransactionSelection::GetNextTransaction

`bool **GetNextTransaction**(MonetaryTransactionData &theData)`

Gets the next MonetaryTransactionData record from the set of rows returned from EdgCapture.

Parameters:

`&theData` - Reference to a MonetaryTransactionData object.

TransactionSelection::GetNumberRowsFound

`int **GetNumberRowsFound**(void)`

Gets the number of rows found for this directory selection.

TransactionSelection::GetNumberRowsReturned

`int **GetNumberRowsReturned**(void)`

Gets the number of rows returned for this directory.

TransactionSelection::SetNumberRowsRequested

`void **SetNumberRowsRequested**(int iRows)`

Sets the value for the maximum number of rows to be returned at one time.

Parameters:

`iRows` - The maximum number of rows to return.

TransactionSelection::SetSelectionProcessSource

`void **SetSelectionProcessSource**(const char *pztProcessSource)`

Sets process source to limit selection. Process source is a client specified string used to tie together transactions from one or more points of origin or one or more users. Process source optionally may be specified when logging onto the EdgCapture from the ECCO API. Then it may be used to group together transactions sharing a process source into a common directory selection.

Parameters:

`pztProcessSource` - null terminated string holding the process source

TransactionSelection::SetSelectionAccountNumber

`void **SetSelectionAccountNumber**(const char *pztAccountNumber)`

Sets the value of the account number to limit selection. Only one account number may be used for selection.

Parameters:

`pztAccountNumber` - null terminated string holding the account number.

TransactionSelection::SetSelectionBeginDate

```
void SetSelectionBeginDate(const char *pztMonth,
               const char *pztDay,
               const char *pztYear)
```

Sets the begin date for Date Range selection. Setting a begin date sets the date option to DATE_OP_DATERANGE.

Parameters:

`pztMonth` - Month of the year (01 - 12)

`pztDay` - Day of the month (01 - 31)

`pztYear` - Four digit year

TransactionSelection::SetSelectionClientUserName

```
void SetSelectionClientUserName(const char *pztClientUserName)
```

Sets one user logon name to limit selection.

Parameters:

`pztClientUserName` - null terminated string with logon name.

TransactionSelection::SetSelectionClientUserName

```
void SetSelectionClientUserName(const char *paztClientUserName[],
               int iCount)
```

Sets one or more client user logon names to limit selection.

Parameters:

`paztClientUserName` - array of null terminated strings containing logon names for selection

`iCount` - count of strings in the array

TransactionSelection::SetSelectionDateOption

```
void SetSelectionDateOption(int iDateOption)
```

Set a date option for selection. The date options are: DATE_OP_TODAY, DATE_OP_YESTERDAY, DATE_OP_THISWEEK, DATE_OP_LASTWEEK, DATE_OP_DATERANGE

Parameters:

`iDateOption` - Date option as available from ECCOClient.

TransactionSelection::SetSelectionEndDate

```
void SetSelectionEndDate(const char *pztMonth,
               const char *pztDay,
               const char *pztYear)
```

Sets the end date for Date Range selection. Setting an end date sets the date option to DATE_OP_DATERANGE.

Parameters:

`pztMonth` - null terminated string holding Month of the year (01 - 12)

`pztDay` - null terminated string holding Day of the month (01 - 31)

`pztYear` - null terminated string holding four digit year

TransactionSelection::SetSelectionName

```
void SetSelectionName(const char *pztName)
```

Sets the value of the customer last name or company name to limit selection. Only one name may be used to limit selection.

Parameters:

`pztName` - null terminated string holding name.

TransactionSelection::SetSelectionMerchantAndOep

```
void SetSelectionMerchantAndOep(const char *pztmerchantId,
               const char *pztOepId = NULL)
```

Set a Merchant and one of its OEPs for selection. If OEP is empty, then all oeps for the merchant are selected. Multiple Merchant/OEP selections can be made by calling this method. Neither parameter can be null.

Note: Each call is OR'd together with previous calls

Examples:

setSelectionMerchantAndOEP ('0', 'A')

Results in the selection: Merchant = '0' AND OEP = 'A'

setSelectionMerchantAndOEP ('0', '')

Results in the selection: Merchant = '0'

setSelectionMerchantAndOEP ('0', 'A')

setSelectionMerchantAndOEP ('1', 'B')

Results int he selection: (Merchant = '0' AND OEP = 'A') OR (Merchant = '1' AND OEP = 'B')

Parameters:

`pztMerchantId` - Null terminated string containing merchant id for selection

`pztOepId` - Null terminated string containing OEP id for selection

TransactionSelection::SetSelectionPaymentType

```
void SetSelectionPaymentType(const char *pztPaymentType)
```

Sets a payment type to limit selection. The ECCO constant payment types are: PAY_AMERICAN_EXPRESS, PAY_CARTE_BLANCHE, PAY_CASH, PAY_CHECK_DRAFT,PAY_DINERS_CLUB, PAY_DIRECT_DEBIT PAY_DISCOVER,PAY_MASTER_CARD, PAY_PAPER_CHECK, PAY_VISA.

Parameters:

`pztPaymentType` - Null terminated string containing payment type for selection

TransactionSelection::SetSelectionPaymentType

```
void SetSelectionPaymentType(const char *paztPaymentType[],
               int iCount)
```

Sets one or more payment types to limit selection. The ECCO constant payment types are:
PAY_AMERICAN_EXPRESS, PAY_CARTE_BLANCHE, PAY_CASH,
PAY_CHECK_DRAFT,PAY_DINERS_CLUB, PAY_DIRECT_DEBIT
PAY_DISCOVER,PAY_MASTER_CARD, PAY_PAPER_CHECK, PAY_VISA

Parameters:

`paztPaymentType` - array of null terminated strings containing payment types for selection

`iCount` - count of strings in the array

TransactionSelection::SetSelectionQueueName

```
void SetSelectionQueueName(const char *pztQueueName)
```

Sets a queue name to limit selection. The ECCO constant queue names are:
QUEUE_AUTHORIZED, QUEUE_CAPTURED, QUEUE_DECLINED, QUEUE_ERROR,
QUEUE_HELD, QUEUE_INPUT, QUEUE_MARKED_FOR_CAPTURE,
QUEUE_RECORDED_ONLY, QUEUE_UNKNOWN_DISPOSITION, QUEUE_VOIDED.

Parameters:

`pztQueueName` - Null terminated string containing queue name for selection

TransactionSelection::SetSelectionQueueName

```
void SetSelectionQueueName(const char *paztQueueName[], int iCount)
```

Sets one or more queue name to limit selection. The ECCO constant queue names are:
QUEUE_AUTHORIZED, QUEUE_CAPTURED, QUEUE_DECLINED, QUEUE_ERROR,
QUEUE_HELD, QUEUE_INPUT, QUEUE_MARKED_FOR_CAPTURE,
QUEUE_RECORDED_ONLY, QUEUE_UNKNOWN_DISPOSITION, QUEUE_VOIDED.

Parameters:

`paztQueueName` - array of null terminated strings containing queue names for selection

`iCount` - count of strings in the array

TransactionSelection::SetSelectionTelephoneNumber

```
void SetSelectionTelephoneNumber(const char *pztPhoneNumber)
```

Sets the value of the telephone number to limit selection. Only one phone number may be used for selection.

Parameters:

`pztPhoneNumber` - Null terminated string containing phone number for selection

TransactionSelection::SetSelectionTransactionType

```
void SetSelectionTransactionType(const char *pzTransactionType)
```

Sets a transaction type to limit selection. The ECCO constant transaction types are: TRAN_AUTHORIZE, TRAN_CAPTURE_HELD_TRANSACTION, TRAN_CREDIT, TRAN_DECLINE_HELD_TRANSACTION, TRAN_MARKED_FOR_CAPTURE, TRAN_RECORD_ONLY, TRAN_VOID.

Parameters:

`pztTransactionId` - Fixed string from ECCOClient that identifies the transaction type.

TransactionSelection::SetSelectionTransactionType

```
void SetSelectionTransactionType(const char *paztTransactionType[],
              int iCount)
```

Sets one or more transaction types to limit selection. The ECCO constant transaction types are: TRAN_AUTHORIZE, TRAN_CAPTURE_HELD_TRANSACTION, TRAN_CREDIT, TRAN_DECLINE_HELD_TRANSACTION, TRAN_MARKED_FOR_CAPTURE, TRAN_RECORD_ONLY, TRAN_VOID.

Parameters:

`paztTransactionType` - array of null terminated strings containing transaction types for selection

`iCount` - count of strings in the array

TransactionSelection::SetSelectionUserData1

```
void SetSelectionUserData1(const char *pztUserData)
```

Sets the value of the 40 character UserData1 field to limit selection. Only one UserData1 may be used to limit selection.

Parameters:

`pztUserData` - Null terminated string containing User Data for selection

TransactionSelection::SetSelectionUserData10

```
void SetSelectionUserData10(const char *pztUserData)
```

Sets the value of the 40 character UserData10 field to limit selection. Only one UserData10 may be used to limit selection.

Parameters:

`pztUserData` - Null terminated string containing User Data for selection

TransactionSelection::SetSelectionUserData2

```
void SetSelectionUserData2(const char *pztUserData)
```

Sets the value of the 40 character UserData2 field to limit selection. Only one UserData2 may be used to limit selection.

Parameters:

`pztUserData` - Null terminated string containing User Data for selection

TransactionSelection::SetSelectionUserData3

```
void SetSelectionUserData3(const char *pztUserData)
```

Sets the value of the 40 character userData3 field to limit selection. Only one userData3 may be used to limit selection.

Parameters:

`pztUserData` - Null terminated string containing User Data for selection

TransactionSelection::SetSelectionUserData4

```
void SetSelectionUserData4(const char *pztUserData)
```

Sets the value of the 40 character UserData4 field to limit selection. Only one UserData4 may be used to limit selection.

Parameters:

`pztUserData` - Null terminated string containing User Data for selection

TransactionSelection::SetSelectionUserData5

```
void SetSelectionUserData5(const char *pztUserData)
```

Sets the value of the 40 character UserData5 field to limit selection. Only one UserData5 may be used to limit selection.

Parameters:

`pztUserData` - Null terminated string containing User Data for selection

TransactionSelection::SetSelectionUserData6

```
void SetSelectionUserData6(const char *pztUserData)
```

Sets the value of the 40 character UserData6 field to limit selection. Only one UserData6 may be used to limit selection.

Parameters:

`pztUserData` - Null terminated string containing User Data for selection

TransactionSelection::SetSelectionUserData7

```
void SetSelectionUserData7(const char *pztUserData)
```

Sets the value of the 40 character UserData7 field to limit selection. Only one UserData7 may be used to limit selection.

Parameters:

`pztUserData` - Null terminated string containing User Data for selection

TransactionSelection::SetSelectionUserData8

`void SetSelectionUserData8(const char *pztUserData)`

Sets the value of the 40 character UserData8 field to limit selection. Only one UserData8 may be used to limit selection.

Parameters:
`pztUserData` - Null terminated string containing User Data for selection

TransactionSelection::SetSelectionUserData9

`void SetSelectionUserData9(const char *pztUserData)`

Sets the value of the 40 character UserData9 field to limit selection. Only one UserData9 may be used to limit selection.

Parameters:
`pztUserData` - Null terminated string containing User Data for selection

The ECCO PHP API

This section describes the ECCO PHP API. The ECCO PHP API utilizes the open source PHP/Java Bridge project (http://php-java-bridge.sourceforge.net/pjb/) using the pure PHP implementation to communicate with the Edgil ECCO Java API. See the READMEeccoPHP.txt file for more details. This section includes short code samples illustrating basic functionality.

Using the PHP API

This section presents more detailed examples of how to process transactions and retrieve information from EdgCapture using the PHP wrapper for the Java API.

Setting up the ECCO.properties file

The ECCO API is delivered with a properties file that is set to use a “Fake” connection to EdgCapture. In “Fake” mode, messages are never actually transmitted to the EdgCapture server. All responses are generated by the java API itself.

When you are ready to submit a transaction to EdgCapture, you must edit the ECCO.properties file to change the connection type to “Ssl” and to specify the connection information for the EdgCapture server. Note that this parameter is case-sensitive and must be entered as documented here with an initial upper-case character.

See “Appendix D: Editing Properties Files” on page 199 for a complete description of the Properties file and the connection information for the EdgCapture server.

The ECCO.properties file also controls logging from the API. For information on logging, see “Logging during development” on page 25.

Connecting to the PHP/Java Bridge

One must make sure the require_once() method specifies the correct ip and port pointing to where the PHP/Java Bridge is running.

```
define("JAVA_HOSTS", "127.0.0.1:2345");

if(!(@include_once("java/Java.inc")))
   require_once("Java.inc");
```

Processing a transaction

The procedure here covers a complete sale transaction, consisting of an authorization request and a mark for capture request. It also explains how to obtain a unique transaction identifier from EdgCapture for an application that does not produce its own identifiers.

Certification

The secure connection between the application and the EdgCapture server is established at startup of the application and before any transactions are submitted.

The ECCOClient method CertifyECCO currently uses edgilca.keystore as described in "Appendix D: Editing Properties Files" on page 199.

Create an ECCOClient object and call the ECCOClient method *certifyECCO* passing the required passwords and certificate alias to certify the client and server. Edgil recommends that the passwords be entered at startup by an actual user, instead of reading them from a configuration file, so that they are not stored as clear text anywhere on the system.

The first parameter for CertifyECCO is the file password for edgilca.keystore, the others are placeholders for future development.

```
// create ECCOClient object
$ECCOClient = new Java("com.edgil.ecco.eccoapi.ECCOClient",
"PHPTestECCO","c:\\edgil\\java\\ECCO.properties");

// certifyECCO
$status = $ECCOClient->certifyECCO($pass, $pass, $alias, "ECCO");
```

Creating the connection

In order to submit a transaction or an information request to EdgCapture, the application must establish a connection to the server by creating an ECCO client.

Create the ECCOClient object.

```
// create ECCOClient object
$ECCOClient = new Java("com.edgil.ecco.eccoapi.ECCOClient",
"PHPTestECCO","c:\\edgil\\java\\ECCO.properties");
```

Call the ECCOClient method *logon,* passing logon and password as parameters for manual processing. Note that the logon and password belong to the application and can be used for any connection. The logon method opens a secure socket, certifying client and server, and sends the logon message to EdgCapture.

```
// Log onto the EdgCapture Server specifying manual processing
$status = $ECCOClient->logon("myLogon","myPassword");
```

Test the results of the request. There are a few reasons for failure, including invalid logon or password, security errors, or connection problems.

Creating and populating the monetary transaction data

Create a *MonetaryTransactionData* object.

```
// create MonetaryTransactionData object
$myTransactionData = new Java(
"com.edgil.ecco.eccoapi.MonenaryTransactionData");
```

Set the data fields in the *MonetaryTransactionData* object. MonetaryTransactionData provides a set method for each individual data field, as well as methods to set several fields at once.

EdgCapture requires a unique transaction identifier for each transaction within a Merchant and OEP. The required fields for an authorization or credit request include:

Data	***Method***	***Description***
MerchantId	setMerchantId	Merchant identifier
OEPId	setOEPId	OEP identifier
Transaction Id	setTransactionId	Unique transaction identifier
Amount	setAmount	Amount of the sale or credit in format *nnnn.nn*; must be positive for all transaction types.
Payment information	setCreditCard setCheckDraft setDirectDebit setToken	Required credit card data includes AccountNumber and ExpirationDate. Required check draft and direct debit data includes ABANumber, AccountNumber, and PaymentDesignator. Token requires a valid CDM token
ECI	setECI	Electronic commerce indicator that describes the source of the transaction. For web applications, the value is generally 6, indicating that the customer's account data was transmitted using SSL.

See "Appendix C: Monetary Transaction Data" on page 195 for a complete description of the data fields. All data fields should be specified as strings.

The following example assumes that the application has created its own transaction id. It sets some data fields individually and others in logical groups:

```
$myTransactionData->setMerchantId("0");
$myTransactionData->setOEPId("B");
$myTransactionData->setTransactionId("mN23d88");
$myTransactionData->setAmount("95.40");
$myTransactionData->setAccountNumber("9999999999999999");
$myTransactionData->setExpirationDate("10","09");
$myTransactionData->setECI("6");
$myTransactionData->setFraudSecurityValue("123");
$myTransactionData->setAddress("99 Wayward Lane","Boston", "MA",
"01776");
$myTransactionData->setName("Jane","Smith");
$myTransactionData->setUserData1("Classified");
$myTransactionData->setUserData2("auto");
$myTransactionData->setUserData3("daily morning,weekly shopper");
$myTransactionData->setUserData4("by Alice");
$myTransactionData->setUserData6("Jan 5 2007");
```

Submitting the transaction to EdgCapture

Call the ECCOClient method *requestAuthorization* passing the previously created MonetaryTransactionData as a parameter.

```
$status = $ECCOClient->requestAuthorization($myTransactionData);
```

Processing the results

Test the return status from *requestAuthorization* to determine whether the transaction request has succeeded. The MonetaryTransactionData object will have all the information returned from EdgCapture, including the authorization code and other processing information for a successful transaction. If the request fails, the fields in the MonetaryTransactionData object remain as they were set before sending the request. If EdgCapture has recorded the transaction, it returns the transaction status, the current queue and other information.

SUCCESS

On ECCO_SUCCESS, the application should record the results, including MerchantId, OEPId, and TransactionId. For a successfully authorized transaction, the application must mark the transaction for capture to notify EdgCapture that the transaction is complete from the application side. If no mark for capture request is sent, the transaction will not be settled. See "Sending a mark for capture request" on page 160 for a description of this process.

To retrieve the authorization code, the current EdgCapture queue, or previously set transaction data, use the MonetaryTransactionData's get methods. For example:

```
$authCode = $myTranactionData->getAuthorizationCode();
```

FAILURE

Check the return status against the codes in ECCO.h. See "Appendix B: ECCO Status Codes" on page 189 for more information.

The type of failure determines the action needed. For data entry errors and declined credit cards, the customer may be asked to reenter information or another account number. The transaction can then be resubmitted using the same connection and Transaction Id.

```
// define ECCOStatusCodes object for ECCO Constants...
$ECCOStatusCodes = new Java("com.edgil.ecco.eccoapi.ECCOStatusCodes");

switch ($status)
{
    case $ECCOStatusCodes->COMPROBLEM:
        //tell the customer to come back later
        break;
    case $ECCOStatusCodes->INVALIDACCOUNTNUMBER:
        //ask the customer to check it or provide another account
        break;
}
```

If you want to display the message text, use the ECCOClient method *getStatusMessage* to retrieve it.

Sending a mark for capture request

The current MonetaryTransaction object can also be used to send the request to mark the transaction for capture.

After the application has ensured that the product has been shipped or delivered to the customer, the successfully authorized transaction must be marked for capture. If the MonetaryTransactionData object reporting the successful authorization still exists, it contains all the necessary information, and the application can use it for the request by calling the ECCOClient's *requestMarkForCapture* method.

```
// mark for capture
$status = $ECCOClient->requestMarkForCapture($myTransactionData);
```

If the application must send the mark for capture message at a later time, follow the steps in the previous section to create a new *ECCOClient* connection and a new *MonetaryTransactionData* object. Set the MerchantId, OEPId, and TransactionId to the values stored for this transaction, and call *requestMarkForCapture*, passing the MonetaryTransactionData as a parameter.

```
$myTransactionData->setMerchantId("0");
$myTransactionData->setOEPId("B");
$myTransactionData->setTransactionId("mN23d88");
$status = $ECCOClient->requestMarkForCapture($myTransactionData);
```

Test the return status for success. See "Appendix B: ECCO Status Codes" on page 189 for information.

Reusing the connection for a subsequent transaction

You can reuse the same client connection for a new transaction or to resubmit a transaction after correcting errors in the data.

When the next transaction is entirely different from the current transaction, clear the MonetaryTransaction object of existing data using *clearAllFields*, which returns all fields to null.

```
$myTransactionData->clearAllFields();
//now set the new ones
```

When a transaction request needs to be resent because of a minor error or when a new transaction is to be performed for the same customer, you do not need to clear the existing data before resetting the incorrect fields. In this example, the customer has entered a new account number:

```
$myTransactionData->setAccountNumber("44444444444444");
```

To continue, submit the transaction request to EdgCapture.

Cleaning up

After all messages have been sent using the current ECCOClient, clean up and disconnect by calling the ECCOClient method *logoff*.

Retrieving transaction data for a single transaction

Each transaction is uniquely identified in the EdgCapture database by its Transaction Key, made up of the Merchant Id, the OEP Id, and the Transaction Id. To retrieve the data for any transaction, the application must have this information available. Note that you must perform a transaction selection in order to void a previously submitted transaction.

If no connection exists, create an ECCOClient and log on, as described in the previous section. Otherwise, use the existing ECCOClient connection.

Create a *TransactionSelection*.

```
$mySelection = new Java("com.edgil.ecco.eccoapi.TransactionSelection");
```

Use the TransactionSelection's *setSelectionTransactionKey* method to set the required Merchant Id, OEP Id and TransactionId that uniquely identify the transaction. You can also set the fields individually.

```
$mySelection->setSelectionTransactionKey("0","B","BE345");
```

Call the ECCOClient method *requestDirectory* passing the previously created TransactionSelection as a parameter.

```
$status = $ECCOClient->requestDirectory($mySelection);
```

Test the return status for success. There are few reasons to fail, although it is possible if the application requests a Merchant or OEP to which it does not have access or if the requested transaction is not in the database.

If the request is successful, access the returned MonetaryTransactionData object.

```
if ($status == $ECCOStatusCodes->SUCCESS)
{
   //pick up the data
   $myTransaction = $mySelection->getNextTransaction()
   {
      //use get methods to retrieve data
}
```

}Use MonetaryTransactionData's get methods to access the data. In this example, the application checks the queue to determine whether the transaction has been authorized.

```
$myQueue = $myTransactionData->getQueueName();
```

To retrieve the data for another transaction using the same ECCOClient connection, reset the selection criteria and call *requestDirectory* again.

To close the connection, call the ECCOClient's *logoff* method.

Voiding a transaction

A void transaction nullifies a previous sale or credit transaction that has not yet been captured. A void can only be entered on the same day during the same settlement period as the transaction it is meant to offset. Voids are used to correct a mistake, such as an incorrect amount or card number, or to cancel a transaction submitted in error.

To void a transaction, request the transaction information as described in the previous section. Using the returned MonetaryTransactionData object, send the void request.

```
$status = $ECCOClient->requestVoid($myTransactionData);
```

Retrieving a directory of transactions

If your site has purchased the directory option, an application can use TransactionSelection to retrieve a group of transactions.

Create the *TransactionSelection* object.

```
$mySelection = new Java("com.edgil.ecco.eccoapi.TransactionSelection");
```

Specify the number of transactions to return by setting a row count. Each row corresponds to the data for a single transaction. This example requests 20 transactions:

```
$mySelection->setNumberRowsRequested(20);
```

Use TransactionSelection's set methods to specify the selection criteria. The selection criteria are defined in the ECCOClient. This example requests all Visa transactions from yesterday that are in the captured queue, that is, all Visa transactions that were included in the previous night's settlement batch.

```
$mySelection->setSelectionPaymentType(ECCOClient.PAY_VISA);
$mySelection->setSelectionQueue(ECCOClient.QUEUE_CAPTURED);
$mySelection->setSelectionDateOptions(
ECCOClient.DATE_OP_YESTERDAY);
```

Send the request to EdgCapture using the ECCOClient's *requestDirectory* method.

```
$status = $ECCOClient->requestDirectory($mySelection);
```

If the request is successful, access the MonetaryTransactionData objects. Use *getNumberRowsReturned* to determine how many transactions were returned. Note that the maximum number of rows for the current example is 20, as specified previously. To determine how many rows the query actually retrieved from the EdgCapture database, use TransactionSelection's *getNumberRowsFound* method.

```
// create MonetaryTransactionData object
$myTransactionData = new Java(
"com.edgil.ecco.eccoapi.MonetaryTransactionData");


if ($status = $ECCOStatusCodes->SUCCESS)
{
   MonetaryTransactionData myTransactionData;

   $numberOfRows = $mySelection->getNumberRowsReturned();
   for ($i = 0; $i < $numberOfRows; $i++)
   {
      $myTransactionData = $mySelection->getNextTransaction();
   }
}
```

Use the MonetaryTransactionData's get methods to access the data from individual fields.

Updating User Data for Transactions

Transactions have user data associated with them. These fields are labelled User1 through User10, plus the notes field. These fields are used to store additional data in EdgCapture for reference by the system sending transactions. For example, a system may use a user field to link EdgCapture transactions to it's own order by storing the order id in a user field.

The update user data request allows the updating of all user data fields plus the notes field. The data is sent via a MonetaryTransactionData object. This object may be returned via a transaction selection, re-used from a previously submitted transaction, or generated from scratch.

requestUpdateUserData will overwrite the existing user data for ALL of the following fields in the associated MonetaryTransactionData object: UserData1 through UserData10 and Notes. All other fields will remain unchanged.

Each transaction is uniquely identified in the EdgCapture database by its TransactionKey, made up of the Merchant Id, the OEP Id, and the Transaction Id. To update the user data for any transaction, the application must have this information available so that the transaction to be updated can be uniquely identified.

If no connection exists, create an ECCOClient and log on, as described in the previous sections. Otherwise, use the existing ECCOClient connection.

You could use a previously used MonetaryTransactionData object or create and populate one with the appropriate data in the transaction key, user fields, and notes field. Use the MonetaryTransactionData's *setTransactionKey* method to set the required MerchantId, OEPId and TransactionId that uniquely identify the transaction. Alternatively, you can also set the fields individually using the setters for TransactionId, MerchantId, and OEPId:

```
$myUpdate->setTransactionKey("0","B","BE345") ;
```

Many times you will be re-using a MonetaryTransactionData object that was returned from a transaction selection or used to process a transaction. In this case you only need to update the user fields and notes fields as needed.

```
$status = $myTransactionData->setUserData1("data") ;
```

Once the data is ready, call the ECCOClient method *requestUpdateUserData*:

```
$status = $ECCOClient->requestUpdateUserData(
$myTransactionData) ;
```

Test the return status for success. If the request is successful, then the specified transaction will have the user data and notes field updated according to what was set in the MonetaryTransactionData.

To close the connection, call the ECCOClient's logoff method.

Cardholder Data Management CDM

Cardholder data can be managed via the CDM requests provided by the ECCOClient class. The user creates a CardholderData object, then uses that object to send CDM requests and get replies.

ECCOClient provides three requests for managing cardholder data. requestCreateToken accepts cardholder data, and returns a token that can then be stored and used to reference the cardholder data. requestGetCardholderData accepts a token and returns the related cardholder data. requestUpdateCardholderData will update the data referenced by a token with new data sent in the request. For example, requestUpdateCardholderData may be used to change the expiration date on an account.

The rest of this section contains a general description of each of the CDM methods available. For detailed syntax, return values, and parameters, see the sections below related to your implementation language.

CDM requests use the CardholderData object. Below is an example of the population of a CardholderData object.

```
$data->setAccountNumber( "4012888888881881" ) ;
$data->setAddress ( "12 Pine Street", "Helton", "NC", "12345" ) ;
$data->setExpirationDate ( "10", "12" ) ;
$data->setName ( "Bebe", "Bekila" ) ;
$data->setTelephone ( "222-222-2222" ) ;
```

Creating Tokens

requestCreateToken is a request that allows the client to store cardholder data in the Edgcapture data base for later use to process transactions. CardholderData objects are sent, and tokens are returned.

The requestCreateToken method is called through the ECCOClient object.

```
 $status = $client->requestCreateToken ( $data, false ) ;
```

Upon return, getToken can then be called to retrieve the created token. If the cardholder data already exists, then a previously existing token will be returned that matches the data, and the return status will be CARDHOLDER_DATA_ALREADY_EXISTS. The token related to that cardholder data will be returned. If new cardholder data is created the status will be SUCCESS.

Retrieving Cardholder Data

requestGetCardHolderData will return the cardholder data for a token. The token must be set in the data.

```
$data->setToken (“123456789”) ;
```

Then the request can be made.

```
$status = $client->requestGetCardholderData ($data)
```

If the token is not found, the return status will be INVALID_TOKEN. Otherwise, the return status will be SUCCESS or some other error.

Updating Cardholder Data

CardHolder Data Management allows the client to update existing cardholder data by sending a token and updated data. The account number can not be updated. To insert a new account number you must use the requestCreateToken method.

After the update, the token will refer to the updated cardholder data. The previous data will remain in the database since it may be linked with transactions. When retrieving a directory of transactions using a token using the setSelectionToken method, the returned transactions will include those linked to the current cardholder data for that token, and all previous versions, since they all relate to the same account number.

When the update cardholder data request is processed, all data fields in the existing cardholder data are overwritten with the new data. Therefore, the client will typically read the existing data, change desired fields, and submit the cardholder data for update. The following example shows this procedure.

Read the data for the token.

```
$data->clearAllFields() ;
$data->setToken ( $tokenForValidCD ) ;
if ( ( $status = $client->requestGetCardholderData ( $data ) ) !=
   $ECCOStatusCodes->SUCCESS )
   {error processing}
```

Modify desired fields.

```
$data->setAddress ( "Updated Address", "Updated City", "RI",
   "55555" ) ;
```

Then request the update.

```
 if ( ( $status = $client->requestUpdateCardholderData ( $data,
   false ) ) !=
   $ECCOStatusCodes->SUCCESS )
{error processing}
```

Debugging a PHP application

The PHP API is based upon the underlying java implementation. Debugging can involve either the interface between the PHP/Java Bridge and the ECCO Java API or the application code.

Debugging the PHP/Java Bridge

The PHP/Java Bridge log file is available if the PHP/Java Bridge is started specifying a log file name. One can also specify a log level as follows (both of these arguments are optional and the default Log Level is 3):

```
0: log off
1: log fatal
2: log messages/exceptions
3: log verbose
4: log debug
5: log method invocations
```

So one would set the Log Level to 4 or 5 for debugging purposes.

Note: The logging output will also be sent to the console if the PHP/Java Bridge is started from the command line and no Log filename is specified.

Note: Also, for production, one would want to either disable this logging or make sure the Log Level is no higher than level 3.

PHP/Java Bridge Log file Example

The following is an example of the PHP/Java Bridge Log file using a log level of 4.

```
Oct 31 14:21:04 JavaBridge INFO : VM                 : 1.6.0_02@http://java.sun.com/
Oct 31 14:21:04 JavaBridge INFO : JavaBridge version             : 4.2.2
Oct 31 14:21:04 JavaBridge INFO : logFile            : javabridge.log
Oct 31 14:21:04 JavaBridge INFO : default logLevel   : 4
Oct 31 14:21:04 JavaBridge INFO : socket             : SERVLET_LOCAL:2345
Oct 31 14:21:04 JavaBridge INFO : java.ext.dirs      : C:\Program
    Files\Java\jre1.6.0_02\lib\ext;C:\WINDOWS\Sun\Java\lib\ext
Oct 31 14:21:04 JavaBridge INFO : php.java.bridge.base: C:\Documents and Set-
    tings\ddemers.EDGNET
Oct 31 14:21:04 JavaBridge INFO : extra library dir   : C:\Documents and Set-
    tings\ddemers.EDGNET\lib
Oct 31 14:21:04 JavaBridge INFO : thread pool size    : 20
Oct 31 14:21:04 JavaBridge INFO : JavaBridgeRunner started on port INET_LOCAL:2345
Oct 31 14:21:12 JavaBridge DEBUG: Socket connection accepted
Oct 31 14:21:12 JavaBridge DEBUG: Starting HTTP server thread from thread pool
Oct 31 14:21:12 JavaBridge DEBUG: Socket connection accepted
Oct 31 14:21:12 JavaBridge DEBUG: Starting HTTP server thread from thread pool
Oct 31 14:21:12 JavaBridge DEBUG: created new bridge: php.java.bridge.Jav-
    aBridge@ab50cd
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@1c5c1 --> <Y p="1" v="0" m="updateJarLi-
    braryPath" >
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@1c5c1 --> <S v="c:\edgil\java\" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@1c5c1 --> <S v="./" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@1c5c1 --> </Y>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@1c5c1  <-- <V />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@1c5c1 re-directing to port# Socket:9267
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@1c5c1 waiting for context: 1@
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <K p="1"
    v="com.edgil.ecco.eccoapi.ECCOStatusCodes" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </K>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <O v="1"
    m="com.edgil.ecco.eccoapi.ECCOStatusCodes" p="O"/>
```

```
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <K p="1"
    v="com.edgil.ecco.eccoapi.ECCOClient" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="PHPTestECCO" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="c:\edgil\java\ECCO.prop-
    erties" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </K>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <O v="2"
    m="com.edgil.ecco.eccoapi.ECCOClient" p="O"/>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <Y p="1" v="2" m="getStatusMes-
    sage" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </Y>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <O v="3" m="java.lang.String"
    p="O"/>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <Y p="1" v="0" m="Object-
    ToString" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <O v="3" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </Y>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <S v="TmV2ZXIgZGlkIGxvZ29u"/>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <U v="3" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <Y p="1" v="2" m="certifyECCO" >
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <X t="H" >
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <P t="N" v="0" >
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="C" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </P>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <P t="N" v="1" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="h" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </P>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <P t="N" v="2" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="a" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </P>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <P t="N" v="3" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="n" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </P>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <P t="N" v="4" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="g" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </P>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <P t="N" v="5" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="e" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </P>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <P t="N" v="6" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="I" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </P>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <P t="N" v="7" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="t" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </P>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </X>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <X t="H" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <P t="N" v="0" >
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="C" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </P>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <P t="N" v="1" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="h" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </P>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <P t="N" v="2" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="a" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </P>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <P t="N" v="3" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="n" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </P>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <P t="N" v="4" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="g" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </P>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <P t="N" v="5" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="e" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </P>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <P t="N" v="6" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="I" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </P>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <P t="N" v="7" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="t" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </P>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </X>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <X t="H" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <P t="N" v="0" >
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="e" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </P>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <P t="N" v="1" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="d" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </P>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <P t="N" v="2" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="g" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </P>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <P t="N" v="3" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="i" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </P>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <P t="N" v="4" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="l" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </P>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <P t="N" v="5" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="c" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </P>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <P t="N" v="6" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="a" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </P>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </X>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="ECCO" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </Y>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <L v="0" p="O"/>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <Y p="1" v="2" m="getStatusMes-
    sage" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </Y>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <O v="4" m="java.lang.String"
    p="O"/>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <Y p="1" v="0" m="Object-
    ToString" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <O v="4" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </Y>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <S v="TmV2ZXIgZGlkIGxvZ29u"/>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <U v="4" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <G p="1" v="1" m="SUCCESS" >
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </G>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <L v="0" p="O"/>
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <Y p="1" v="2" m="logon" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="ECCOClient" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="ECCOClient" />
Oct 31 14:21:12 JavaBridge DEBUG: ab50cd@a3bcc1 --> </Y>
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <L v="0" p="O"/>
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <Y p="1" v="2" m="getStatusMes-
    sage" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> </Y>
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <O v="5" m="java.lang.String"
    p="O"/>
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <Y p="1" v="0" m="Object-
    ToString" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <O v="5" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> </Y>
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <S v="U3VjY2Vzcw=="/>
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <U v="5" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <G p="1" v="1" m="SUCCESS" >
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> </G>
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <L v="0" p="O"/>
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <K p="1"
    v="com.edgil.ecco.eccoapi.MonetaryTransactionData" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> </K>
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <O v="6"
    m="com.edgil.ecco.eccoapi.MonetaryTransactionData" p="O"/>
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <Y p="1" v="6" m="setMerchantId"
    />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="0" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> </Y>
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <V />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <Y p="1" v="6" m="setOEPId" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="A" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> </Y>
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <V />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <Y p="1" v="6" m="setTransac-
    tionId" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="test_31-Oct-
    2007.14.21.13-30" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> </Y>
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <V />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <Y p="1" v="6" m="setAmount" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="02.21" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> </Y>
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <V />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <Y p="1" v="6" m="setCreditCard"
```

```
    />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="4012********1881" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="11" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="12" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> </Y>
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <V />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <Y p="1" v="6" m="setECI" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="6" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> </Y>
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <V />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <Y p="1" v="6" m="setFraudSecu-
    rityValue" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="123" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> </Y>
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <V />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <Y p="1" v="6" m="setAddress" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="123 Fortune Drive" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="Billerica" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="MA" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="01824" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> </Y>
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <V />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <Y p="1" v="6" m="setName" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="Jane" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="Smith" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> </Y>
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <V />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <Y p="1" v="6" m="setUserData1"
    />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <S v="UserData1" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> </Y>
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <V />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <Y p="1" v="2" m="requestAutho-
    rization" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> <O v="6" />
Oct 31 14:21:13 JavaBridge DEBUG: ab50cd@a3bcc1 --> </Y>
Oct 31 14:21:15 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <L v="0" p="O"/>
Oct 31 14:21:15 JavaBridge DEBUG: ab50cd@a3bcc1 --> <G p="1" v="1" m="SUCCESS" />
Oct 31 14:21:15 JavaBridge DEBUG: ab50cd@a3bcc1 --> </G>
Oct 31 14:21:15 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <L v="0" p="O"/>
Oct 31 14:21:15 JavaBridge DEBUG: ab50cd@a3bcc1 --> <Y p="1" v="6" m="getAuthoriza-
    tionCode" />
Oct 31 14:21:15 JavaBridge DEBUG: ab50cd@a3bcc1 --> </Y>
Oct 31 14:21:15 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <O v="7" m="java.lang.String"
    p="O"/>
Oct 31 14:21:15 JavaBridge DEBUG: ab50cd@a3bcc1 --> <Y p="1" v="0" m="Object-
    ToString" />
Oct 31 14:21:15 JavaBridge DEBUG: ab50cd@a3bcc1 --> <O v="7" />
Oct 31 14:21:15 JavaBridge DEBUG: ab50cd@a3bcc1 --> </Y>
Oct 31 14:21:15 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <S v="MDcxMDMxRks0Mg=="/>
Oct 31 14:21:15 JavaBridge DEBUG: ab50cd@a3bcc1 --> <Y p="1" v="2" m="requestMark-
    ForCapture" />
Oct 31 14:21:15 JavaBridge DEBUG: ab50cd@a3bcc1 --> <O v="6" />
Oct 31 14:21:15 JavaBridge DEBUG: ab50cd@a3bcc1 --> </Y>
Oct 31 14:21:16 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <L v="0" p="O"/>
Oct 31 14:21:16 JavaBridge DEBUG: ab50cd@a3bcc1 --> <G p="1" v="1" m="SUCCESS" />
Oct 31 14:21:16 JavaBridge DEBUG: ab50cd@a3bcc1 --> </G>
Oct 31 14:21:16 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <L v="0" p="O"/>
Oct 31 14:21:16 JavaBridge DEBUG: ab50cd@a3bcc1 --> <Y p="1" v="2" m="logoff" />
Oct 31 14:21:16 JavaBridge DEBUG: ab50cd@a3bcc1 --> </Y>
Oct 31 14:21:16 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <V />
Oct 31 14:21:16 JavaBridge DEBUG: ab50cd@a3bcc1 --> <G p="1" v="1" m="SUCCESS" />
Oct 31 14:21:16 JavaBridge DEBUG: ab50cd@a3bcc1 --> </G>
Oct 31 14:21:16 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <L v="0" p="O"/>
Oct 31 14:21:16 JavaBridge DEBUG: ab50cd@a3bcc1 --> <U v="7" />
Oct 31 14:21:16 JavaBridge DEBUG: ab50cd@a3bcc1 --> <U v="6" />
Oct 31 14:21:16 JavaBridge DEBUG: ab50cd@a3bcc1 --> <U v="2" />
Oct 31 14:21:16 JavaBridge DEBUG: ab50cd@a3bcc1 --> <U v="1" />
Oct 31 14:21:16 JavaBridge DEBUG: ab50cd@a3bcc1 --> <F p="A" />
Oct 31 14:21:16 JavaBridge DEBUG: contextfactory: finish context called (recycle
    context factory) ContextFactory: Context# 1@, isInitialized: true, isValid:
    true, SimpleContextFactory: class php.java.bridge.http.SimpleContextFactory,
    current loader: java.net.URLClassLoader@c51355
Oct 31 14:21:16 JavaBridge DEBUG: ab50cd@a3bcc1  <-- <F p="A"/>
```

Setting up logging for development (Java API)

The ECCO Java API uses the SimpleSocketServer task to log to a user-specified file, which contains a single day's messages. The file is truncated daily at the first message received after midnight, and the previous day's messages are saved in a file with a date extension. API logging is controlled by an ECCO properties file, whose options may be modified by method calls within the application. For additional information about logging from the application, see "Logging during development" on page 25.

Setting up the SimpleSocketServer

The SimpleSocketServer task must be running in order to log to a file. Use StartSimpleSocketServer.bat to start the task. The delivered SimpleSocketServerECCO.properties writes the log files to c:\edgil\data\ecco. Edit this file to specify a different directory structure or a different java classpath. For information on editing the properties file, see "Appendix D: Editing Properties Files" on page 199.

ECCO.properties

The ECCO.properties file is delivered with logging set to report only fatal errors to the console; this is the default for normal operation. To set up logging for development, you can either edit the default ECCO.properties file or create another properties file thatspecifies logging by uncommenting the logging sections for the output you want.You must also edit the log4j.appender.SocketApp.RemoteHost line to specify either the name or the IP address of the local machine. Here the machine name is "raven":

```
log4j.appender.SocketApp.RemoteHost=raven
```

For information on editing the properties file, see "Appendix D: Editing Properties Files" on page 199. Note that if you use the default ECCO.properties file for development, you must edit it after the debugging process is finished to prevent API logging in a production environment.

Logging from ECCO classes

The ECCO classes may be used from any thread regardless of where they are created. Therefore each instantiation of an ECCO class given a session id to use for logging. When an ECCO class is created and logging is desired, you must specify the ECCO properties file and a common session id.

To use ECCO logging, the application should create its own ECCOLog object after creating the ECCOClient. This example logs the messages "this is a warning" and "starting up my app" (one would replace the hardcode "classname" with the actual class name to record the source) :

```
$ECCOClient = new Java("com.edgil.ecco.eccoapi.ECCOClient",
   "mySession","c:\edgil\data\myPropertiesFile.properties");

$ECCOLog = new Java("com.edgil.ecco.eccoapi.ECCOLog", "class-
   name");
$ECCOLog->setPriority($ECCOLog->WARN);
$ECCOLog->warn("this is a warning");
```

```
$ECCOLog->setPriority($ECCOLog->INFO)
$ECCOLog->info("starting up my app");
```

ECCO PHP Class Reference

Please refer to the ECCO Java API Class Reference section for complete descriptions of the four ECCO classes provided for use by the PHP programmer: ECCOClient, MonetaryTransactionData, TransactionSelection and ECCOLog.

The ECCO .NET API

This section describes the ECCO .NET API. The ECCO .NET API utilizes the open source IKVM.NET project (http://www.ikvm.net/) which is an implementation of Java for Mono and the Microsoft .NET Framework. See the READMEeccoDotNet.txt file for more details. This section includes short code samples illustrating basic functionality.

Using the .NET API

This section presents more detailed examples of how to process transactions and retrieve information from EdgCapture using the .NET wrapper for the Java API.

Setting up the ECCO.properties file

The ECCO API is delivered with a properties file that is set to use a “Fake” connection to EdgCapture. In “Fake” mode, messages are never actually transmitted to the EdgCapture server. All responses are generated by the java API itself.

When you are ready to submit a transaction to EdgCapture, you must edit the ECCO.properties file to change the connection type to “Ssl” and to specify the connection information for the EdgCapture server. Note that this parameter is case-sensitive and must be entered as documented here with an initial upper-case character.

See “Appendix D: Editing Properties Files” on page 199 for a complete description of the Properties file and the connection information for the EdgCapture server.

The ECCO.properties file also controls logging from the API. For information on logging, see “Logging during development” on page 25.

Using the ECCO .NET API

A DLL, EccoDotNetAPI.dll, is provided to use with .NET applications. In addition one must also include some other DLLs. See the READMEeccoDotNet.txt file for details.

Processing a transaction

The procedure here covers a complete sale transaction, consisting of an authorization request and a mark for capture request. It also explains how to obtain a unique transaction identifier from EdgCapture for an application that does not produce its own identifiers.

Certification

The secure connection between the application and the EdgCapture server is established at startup of the application and before any transactions are submitted.

The ECCOClient method CertifyECCO currently uses edgilca.keystore as described in “Appendix D: Editing Properties Files” on page 199.

Create an ECCOClient object and call the ECCOClient method *certifyECCO* passing the required passwords and certificate alias to certify the client and server. Edgil recommends that the passwords be entered at startup by an actual user, instead of reading them from a configuration file, so that they are not stored as clear text anywhere on the system.

The first parameter for CertifyECCO is the file password for edgilca.keystore, the others are placeholders for future development.

```
// create ECCOClent object
com.edgil.ecco.eccoapi.ECCOClient ECCOClient = new ECCOClient
("DotNetTestECCO","c:\\edgil\\java\\ECCO.properties");
```

or (if using “using directive” –> using com.edgil.ecco.eccoapi;)

```
// create ECCOClent object
ECCOClient ECCOClient = new ECCOClient
("DotNetTestECCO","c:\\edgil\\java\\ECCO.properties");

// declare and populate password and alias strings (char arrays)
// . . .

// certifyECCO
int status = ECCOClient.certifyECCO(password, password, alias );
```

Test the results of the certifyECCO request. For all the methods that return a status, one should verify that the call was successful.

Creating the connection

In order to submit a transaction or an information request to EdgCapture, the application must establish a connection to the server by creating an ECCO client.

Create the ECCOClient object.

```
// create ECCOClent object
ECCOClient ECCOClient = new
ECCOClient("DotNetTestECCO","c:\\edgil\\java\\ECCO.properties");

// then call certifyECCO
```

Call the ECCOClient method logon, passing logon and password as parameters for manual processing. Note that the logon and password belong to the application and can be used for any connection. The logon method opens a secure socket, certifying client and server, and sends the logon message to EdgCapture.

```
// logon to the EdgCapture Server
status = ECCOClient.logon("myLogon","myPassword");
```

Test the results of the request. There are a few reasons for failure, including invalidlogon or password, security errors, or connection problems.

Creating and populating the monetary transaction data

Create a *MonetaryTransactionData* object.

```
// create MonetaryTransactionData object
MonetaryTransactionData myTransactionData = new
MonenaryTransactionData();
```

Set the data fields in the *MonetaryTransactionData* object. MonetaryTransactionData provides a set method for each individual data field, as well as methods to set several fields at once.

EdgCapture requires a unique transaction identifier for each transaction within a Merchant and OEP. The required fields for an authorization or credit request include:

Data	***Method***	***Description***
MerchantId	setMerchantId	Merchant identifier
OEPId	setOEPId	OEP identifier
Transaction Id	setTransactionId	Unique transaction identifier
Amount	setAmount	Amount of the sale or credit in format *nnnn.nn*; must be positive for all transaction types.
Payment information	setCreditCard setCheckDraft setDirectDebit setToken	Required credit card data includes AccountNumber and ExpirationDate. Required check draft and direct debit data includes ABANumber, AccountNumber, and PaymentDesignator. Token requires a valid CDM token
ECI	setECI	Electronic commerce indicator that describes the source of the transaction. For web applications, the value is generally 6, indicating that the customer's account data was transmitted using SSL.

See "Appendix C: Monetary Transaction Data" on page 195 for a complete description of the data fields. All data fields should be specified as strings.

The following example assumes that the application has created its own transaction id. It sets some data fields individually and others in logical groups:

```
myTransactionData.setMerchantId("0");
myTransactionData.setOEPId("B");
myTransactionData.setTransactionId("mN23d88");
myTransactionData.setAmount("95.40");
myTransactionData.setAccountNumber("9999999999999999");
myTransactionData.setExpirationDate("10","09");
myTransactionData.setECI("6");
myTransactionData.setFraudSecurityValue("123");
myTransactionData.setAddress("99 Wayward Lane","Boston", "MA",
"01776");
myTransactionData.setName("Jane","Smith");
myTransactionData.setUserData1("Classified");
myTransactionData.setUserData2("auto");
myTransactionData.setUserData3("daily morning,weekly shopper");
myTransactionData.setUserData4("by Alice");
myTransactionData.setUserData6("Jan 5 2007");
```

Submitting the transaction to EdgCapture

Call the ECCOClient method *requestAuthorization* passing the previously created MonetaryTransactionData as a parameter.

```
status = ECCOClient.requestAuthorization(myTransactionData);
```

Processing the results

Test the return status from *requestAuthorization* to determine whether the transaction request has succeeded. The MonetaryTransactionData object will have all the information returned from EdgCapture, including the authorization code and other processing information for a successful transaction. If the request fails, the fields in the MonetaryTransactionData object remain as they were set before sending the request. If EdgCapture has recorded the transaction, it returns the transaction status, the current queue and other information.

SUCCESS

On ECCO_SUCCESS, the application should record the results, including MerchantId, OEPId, and TransactionId. For a successfully authorized transaction, the application must mark the transaction for capture to notify EdgCapture that the transaction is complete from the application side. If no mark for capture request is sent, the transaction will not be settled. See "Sending a mark for capture request" on page 179 for a description of this process.

To retrieve the authorization code, the current EdgCapture queue, or previously set transaction data, use the MonetaryTransactionData's get methods. For example:

```
String authCode = myTranactionData.getAuthorizationCode();
```

FAILURE

Check the return status against the codes in ECCO.h. See "Appendix B: ECCO Status Codes" on page 189 for more information.

The type of failure determines the action needed. For data entry errors and declined credit cards, the customer may be asked to reenter information or another account number. The transaction can then be resubmitted using the same connection and Transaction Id.

```
// define ECCOStatusCodes object for ECCO Constants...
ECCOStatusCodes ECCOStatusCodes = new ECCOStatusCodes();
switch (status)
{
case ECCOStatusCodes.COMPROBLEM:
//tell the customer to come back later
break;
case ECCOStatusCodes.INVALIDACCOUNTNUMBER:
//ask the customer to check it or provide another account
break;
}
```

If you want to display the message text, use the ECCOClient method *getStatusMessage* to retrieve it.

Sending a mark for capture request

The current MonetaryTransaction object can also be used to send the request to mark the transaction for capture.

After the application has ensured that the product has been shipped or delivered to the customer, the successfully authorized transaction must be marked for capture. If the MonetaryTransactionData object reporting the successful authorization still exists, it contains all the necessary information, and the application can use it for the request by calling the ECCOClient's *requestMarkForCapture* method.

```
// mark for capture
status = ECCOClient.requestMarkForCapture(myTransactionData);
```

If the application must send the mark for capture message at a later time, follow the steps in the previous section to create a new *ECCOClient* connection and a new *MonetaryTransactionData* object. Set the MerchantId, OEPId, and TransactionId to the values stored for this transaction, and call *requestMarkForCapture*, passing the MonetaryTransactionData as a parameter.

```
myTransactionData.setMerchantId("0");
myTransactionData.setOEPId("B");
myTransactionData.setTransactionId("mN23d88");
status = $ECCOClient.requestMarkForCapture(myTransactionData);
```

Test the return status for success. See "Appendix B: ECCO Status Codes" on page 189 for information.

Reusing the connection for a subsequent transaction

You can reuse the same client connection for a new transaction or to resubmit a transaction after correcting errors in the data.

When the next transaction is entirely different from the current transaction, clear the MonetaryTransaction object of existing data using *clearAllFields*, which returns all fields to null.

```
myTransactionData.clearAllFields();
//now set the new ones
```

When a transaction request needs to be resent because of a minor error or when a new transaction is to be performed for the same customer, you do not need to clear the existing data before resetting the incorrect fields. In this example, the customer has entered a new account number:

```
myTransactionData.setAccountNumber("4444444444444");
```

To continue, submit the transaction request to EdgCapture.

Cleaning up

After all messages have been sent using the current ECCOClient, clean up and disconnect by calling the ECCOClient method *logoff*.

Retrieving transaction data for a single transaction

Each transaction is uniquely identified in the EdgCapture database by its Transaction Key, made up of the Merchant Id, the OEP Id, and the Transaction Id. To retrieve the data for any transaction, the application must have this information available. Note that you must perform a transaction selection in order to void a previously submitted transaction.

If no connection exists, create an ECCOClient and log on, as described in the previous section. Otherwise, use the existing ECCOClient connection.

Create a *TransactionSelection*.

```
TransactionSlection mySelection = new TransactionSelection();
```

Use the TransactionSelection's *setSelectionTransactionKey* method to set the required Merchant Id, OEP Id and TransactionId that uniquely identify the transaction. You can also set the fields individually.

```
mySelection.setSelectionTransactionKey("0","B","BE345");
```

Call the ECCOClient method *requestDirectory* passing the previously created TransactionSelection as a parameter.

```
status = ECCOClient.requestDirectory(mySelection);
```

Test the return status for success. There are few reasons to fail, although it is possible if the application requests a Merchant or OEP to which it does not have access or if the requested transaction is not in the database.

If the request is successful, access the returned MonetaryTransactionData object.

```
if (status == ECCOStatusCodes.SUCCESS)
{
//pick up the data
myTransaction = mySelection.getNextTransaction()
{
//use get methods to retrieve data
}
```

Use MonetaryTransactionData's get methods to access the data. In this example, the application checks the queue to determine whether the transaction has been authorized.

```
myQueue = myTransactionData.getQueueName();
```

To retrieve the data for another transaction using the same ECCOClient connection, reset the selection criteria and call *requestDirectory* again.

To close the connection, call the ECCOClient's *logoff* method.

Voiding a transaction

A void transaction nullifies a previous sale or credit transaction that has not yet been captured. A void can only be entered on the same day during the same settlement period as the transaction it is meant to offset. Voids are used to correct a mistake, such as an incorrect amount or card number, or to cancel a transaction submitted in error.

To void a transaction, request the transaction information as described in the previous section. Using the returned MonetaryTransactionData object, send the void request.

```
status = ECCOClient.requestVoid(myTransactionData);
```

Retrieving a directory of transactions

If your site has purchased the directory option, an application can use TransactionSelection to retrieve a group of transactions.

Create the *TransactionSelection* object.

```
TransactionSelection mySelection = new TransactionSelection();
```

Specify the number of transactions to return by setting a row count. Each row corresponds to the data for a single transaction. This example requests 20 transactions:

```
mySelection.setNumberRowsRequested(20);
```

Use TransactionSelection's set methods to specify the selection criteria. The selection criteria are defined in the ECCOClient. This example requests all Visa transactions from yesterday that are in the captured queue, that is, all Visa transactions that were included in the previous night's settlement batch.

```
mySelection.setSelectionPaymentType(ECCOClient.PAY_VISA);
mySelection.setSelectionQueue(ECCOClient.QUEUE_CAPTURED);
mySelection.setSelectionDateOptions(
ECCOClient.DATE_OP_YESTERDAY);
```

Send the request to EdgCapture using the ECCOClient's *requestDirectory* method.

```
status = ECCOClient.requestDirectory(mySelection);
```

If the request is successful, access the MonetaryTransactionData objects. Use *getNumberRowsReturned* to determine how many transactions were returned. Note that the maximum number of rows for the current example is 20, as specified previously. To determine how many rows the query actually retrieved from the EdgCapture database, use TransactionSelection's *getNumberRowsFound* method.

```
// create MonetaryTransactionData object
MonetryTransactionData myTransactionData = new
MonetaryTransactionData();

if (status = ECCOStatusCodes.SUCCESS)
{
   MonetaryTransactionData myTransactionData;
   numberOfRows = mySelection.getNumberRowsReturned();
```

```
    for ($i = 0; $i < numberOfRows; $i++)
    {
       myTransactionData = mySelection.getNextTransaction();
    }
}
```

Use the MonetaryTransactionData’s get methods to access the data from individual fields.

Updating User Data for Transactions

Transactions have user data associated with them. These fields are labelled User1 through User10, plus the notes field. These fields are used to store additional data in EdgCapture for reference by the system sending transactions. For example, a system may use a user field to link EdgCapture transactions to it's own order by storing the order id in a user field.

The update user data request allows the updating of all user data fields plus the notes field. The data is sent via a MonetaryTransactionData object. This object may be returned via a transaction selection, re-used from a previously submitted transaction, or generated from scratch.

requestUpdateUserData will overwrite the existing user data for ALL of the following fields in the associated MonetaryTransactionData object: UserData1 through UserData10 and Notes. All other fields will remain unchanged.

Each transaction is uniquely identified in the EdgCapture database by its TransactionKey, made up of the Merchant Id, the OEP Id, and the Transaction Id. To update the user data for any transaction, the application must have this information available so that the transaction to be updated can be uniquely identified.

If no connection exists, create an ECCOClient and log on, as described in the previous sections. Otherwise, use the existing ECCOClient connection.

You could use a previously used MonetaryTransactionData object or create and populate one with the appropriate data in the transaction key, user fields, and notes field. Use the MonetaryTransactionData's *setTransactionKey* method to set the required MerchantId, OEPId and TransactionId that uniquely identify the transaction. Alternatively, you can also set the fields individually using the setters for TransactionId, MerchantId, and OEPId:

```
myUpdate.setTransactionKey("0","B","BE345") ;
```

Many times you will be re-using a MonetaryTransactionData object that was returned from a transaction selection or used to process a transaction. In this case you only need to update the user fields and notes fields as needed.

```
status = myTransactionData.setUserData1(“data”) ;
```

Once the data is ready, call the ECCOClient method *requestUpdateUserData*:

```
status = ECCOClient.requestUpdateUserData(
myTransactionData) ;
```

Test the return status for success. If the request is successful, then the specified transaction will have the user data and notes field updated according to what was set in the MonetaryTransactionData.

To close the connection, call the ECCOClient's logoff method.

Cardholder Data Management CDM

Cardholder data can be managed via the CDM requests provided by the ECCOClient class. The user creates a CardholderData object, then uses that object to send CDM requests and get replies.

ECCOClient provides three requests for managing cardholder data. requestCreateToken accepts cardholder data, and returns a token that can then be stored and used to reference the cardholder data. requestGetCardholderData accepts a token and returns the related cardholder data. requestUpdateCardholderData will update the data referenced by a token with new data sent in the request. For example, requestUpdateCardholderData may be used to change the expiration date on an account.

The rest of this section contains a general description of each of the CDM methods available. For detailed syntax, return values, and parameters, see the sections below related to your implementation language.

CDM requests use the CardholderData object. Below is an example of the population of a CardholderData object.

```
data.setAccountNumber( "4012888888881881" ) ;
data.setAddress ( "12 Pine Street", "Helton", "NC", "12345" ) ;
data.setExpirationDate ( "10", "12" ) ;
data.setName ( "Bebe", "Bekila" ) ;
data.setTelephone ( "222-222-2222" ) ;
```

Creating Tokens

requestCreateToken is a request that allows the client to store cardholder data in the Edgcapture data base for later use to process transactions. CardholderData objects are sent, and tokens are returned.

The requestCreateToken method is called through the ECCOClient object.

```
int status ;
 status = client.requestCreateToken ( data, false ) ;
```

Upon return, getToken can then be called to retrieve the created token. If the cardholder data already exists, then a previously existing token will be returned that matches the data, and the return status will be CARDHOLDER_DATA_ALREADY_EXISTS. The token related to that cardholder data will be returned. If new cardholder data is created the status will be SUCCESS.

Retrieving Cardholder Data

requestGetCardHolderData will return the cardholder data for a token. The token must be set in the data.

```
data.setToken ("123456789") ;
```

Then the request can be made.

```
int status ;
status = client.requestGetCardholderData (data)
```

If the token is not found, the return status will be INVALID_TOKEN. Otherwise, the return status will be SUCCESS or some other error.

Updating Cardholder Data

CardHolder Data Management allows the client to update existing cardholder data by sending a token and updated data. The account number can not be updated. To insert a new account number you must use the requestCreateToken method.

After the update, the token will refer to the updated cardholder data. The previous data will remain in the database since it may be linked with transactions. When retrieving a directory of transactions using a token using the setSelectionToken method, the returned transactions will include those linked to the current cardholder data for that token, and all previous versions, since they all relate to the same account number.

When the update cardholder data request is processed, all data fields in the existing cardholder data are overwritten with the new data. Therefore, the client will typically read the existing data, change desired fields, and submit the cardholder data for update. The following example shows this procedure.

Read the data for the token.

```
data.clearAllFields() ;
data.setToken ( tokenForValidCD ) ;
if ( ( status = client.requestGetCardholderData ( data ) ) !=
   ECCOStatusCodes.SUCCESS )
   {error processing}
```

Modify desired fields.

```
data.setAddress ( "Updated Address", "Updated City", "RI", "55555"
   ) ;
```

Then request the update.

```
 if ( ( status = client.requestUpdateCardholderData ( data, false )
   ) !=
   ECCOStatusCodes.SUCCESS )
   {error processing}
```

Debugging a .NET application

The .Net API is based upon the underlying ECCO java API implementation. So for debugging on'e own code using the .Net API, one can use the IDE such as Visual Studio to debug the application code or one can enable debugging in the ECCO Java API interface to troubleshoot problems that are outside of one's application.

See the following section for information on debugging the ECCO Java API.

Setting up logging for development (Java API)

The ECCO Java API uses the SimpleSocketServer task to log to a user-specified file, which contains a single day's messages. The file is truncated daily at the first message received after midnight, and the previous day's messages are saved in a file with a date extension. API logging is controlled by an ECCO properties file, whose options may be modified by method calls within the application. For additional information about logging from the application, see "Logging during development" on page 25.

Setting up the SimpleSocketServer

The SimpleSocketServer task must be running in order to log to a file. Use StartSimpleSocketServer.bat to start the task. The delivered SimpleSocketServerECCO.properties writes the log files to c:\edgil\data\ecco. Edit this file to specify a different directory structure or a different java classpath. For information on editing the properties file, see "Appendix D: Editing Properties Files" on page 199.

ECCO.properties

The ECCO.properties file is delivered with logging set to report only fatal errors to the console; this is the default for normal operation. To set up logging for development, you can either edit the default ECCO.properties file or create another properties file thatspecifies logging by uncommenting the logging sections for the output you want.You must also edit the log4j.appender.SocketApp.RemoteHost line to specify either the name or the IP address of the local machine. Here the machine name is "raven":

```
log4j.appender.SocketApp.RemoteHost=raven
```

For information on editing the properties file, see "Appendix D: Editing Properties Files" on page 199. Note that if you use the default ECCO.properties file for development, you must edit it after the debugging process is finished to prevent API logging in a production environment.

Logging from ECCO classes

The ECCO classes may be used from any thread regardless of where they are created. Therefore each instantiation of an ECCO class given a session id to use for logging. When an ECCO class is created and logging is desired, you must specify the ECCO properties file and a common session id.

To use ECCO logging, the application should create its own ECCOLog object after creating the ECCOClient. This example logs the messages “this is a warning” and “starting up my app” (one would replace the hardcode “classname” with the actual class name to record the source) :

```
ECCOClient ECCOClient = new ECCOClient
   "mySession","c:\edgil\data\myPropertiesFile.properties");

ECCOLog ECCOLog = new ECCOLog("classname");
ECCOLog.getPriority($ECCOLog->WARN);
ECCOLog.warn("this is a warning");

ECCOLog.setPriority($ECCOLog->INFO)
ECCOLog.info("starting up my app");
```

ECCO .NET Class Reference

Please refer to the ECCO Java API Class Reference section for complete descriptions of the four ECCO classes provided for use by the .NET programmer: ECCOClient, MonetaryTransactionData, TransactionSelection and ECCOLog.

Appendix A: Test data

The delivered ECCO.properties file is set up for testing your application without an actual connection to an EdgCapture server. The following line sets up fake operation:

```
com.edgil.ecco.eccoapi.serverConnectionType=Fake
```

When transactions are submitted using a fake connection, the library returns partially hardcoded results. “Fake” operation allows only credit cards, where the account data consists of account number and expiration date. It checks that the account number starts with either 3 (American Express), 4 (Visa), 5 (Master Card), or 6 (Discover), but does not check the length or the validity of the entire account number. You can manipulate the status returned by setting one of the account numbers in the following table. Note that the status returned depends entirely upon the account number used, regardless of the actual validity of the other data.

Account #	***Status code***	***Status message***
4212000098765437	31	Declined
5499750000000007	30	Call Center
6011000731519850	37	Com Problem
4301790000159206	49	Credit Not Allowed
5334700378609508	68	Invalid State
4085025996546567	69	Invalid ZipCode
5490992902040454	75	Invalid Expiration Date
4777676110022860	113	Expiration Date Missing
4012888888881881	0	Success
Missing or invalid (not beginning with 3, 4, 5, 6)	64	Invalid account number

Appendix B: ECCO Status Codes

The following status codes are contained in the ECCOStatusCodes object. Status codes beginning with INVALID indicate bad data in the request message. For C++, these status strings contained in ECCO.h are preceded by ‘ECCO_’.

String	*Code*	*Description*
SUCCESS	0	Success: The request has been processed successfully.
SERVER_DUPLICATE	3	Server duplicate: The transaction already exists in the database and is not in the Declined or Error queues. This status is returned if you send in the same transaction twice in succession with the same transaction ID.
INVALID_MERCHANTID	5	Invalid Merchant ID: The Merchant ID does not exist.
MERCHANT_CLOSED	7	Merchant in EdgCapture is not available to accept transactions.
SECURITY_ERROR	9	Security error: The client does not have sufficient privilege to perform the specified operation. This is a configuration issue.
DATABASE_ERROR	12	Database error: A fatal error occurred while reading or writing the database. This is indicative of a hardware or software problem.
CALLCENTER	30	Call Center: The transaction was placed into the Held queue of the specified Merchant Account because the credit network requires that the merchant contact the authorization network by telephone in order to obtain an authorization. The transaction may be released for processing by issuing a capture held transaction request.
DECLINED	31	Declined: The Sale, Authorization or Post Authorization transaction was declined and moved to the 'Declined' queue for the specified Merchant Account.
NETWORK_DUPLICATE	32	Network duplicate: Status returned by the credit network if it encounters two transactions with the same amount and account number within a given period.
NETWORK_PROBLEM_RETRY	33	Network problem retry
NO_TRANSACTION_FOUND	36	The transaction identified by the merchant, OEP, and transaction Ids was not found in the EdgCapture database.
COM_PROBLEM	37	Communications problem

String	***Code***	***Description***
INVALID_NETWORK_REPLY	38	Invalid network reply: The credit network returned a reply which is not understood by EdgCapture. This is indicative of a software problem in the credit network or in EdgCapture.
DECLINED_CALLCENTER	41	Declined Call Center: The transaction was automatically declined by EdgCapture after the network issued a call center response. This is an EdgCapture client option.
SYS_CONFIG_ERROR	50	System or configuration error
OVER_MAX_TXN_AMOUNT	51	Transaction discarded: amount too large
INVALID_AMOUNT	61	Invalid amount
INVALID_AUTHORIZATION_CODE	62	Invalid authorization code
INVALID_ACCOUNT_NUMBER	64	Invalid account number; fails algorithm check or otherwise bad.
INVALID_TRACK_DATA	65	Invalid track data
INVALID_ADDRESS	66	Invalid address
INVALID_CITY	67	Invalid city
INVALID_STATE	68	Invalid state
INVALID_ZIPCODE	69	Invalid zip code
INVALID_TELEPHONE_NUMBER	70	Invalid telephone number
INVALID_CLIENT_DATA	72	Invalid client data
INVALID_EXPIRATION_DATE	75	Invalid expiration date; past expiration date or otherwise bad.
INVALID_OEP	89	Invalid OEP does not match an OEP in the database.
INVALID_TRANSACTION_TYPE	90	Invalid transaction type
INVALID_TRANSACTIONID	91	Invalid transaction number
INVALID_ABA_ROUTING_NUMBER	95	Invalid ABA routing number
INVALID_ADPAY_TRANSACTION_TYPE	96	Invalid transaction type for check draft or direct debit
INVALID_ADPAY_PAYMENT_DESIGNATOR	97	Invalid payment designator for check draft or direct debit transaction.
NO_SERVER_CONNECTION	98	No connection to EdgCapture server
INVALID_CARD_TYPE	99	Invalid card type; card not supported by merchant account.
INVALID_PAYMENT_TYPE	101	Invalid payment type; payment type not supported by merchant account
ABA_ROUTING_NUMBER_MISSING	103	ABA routing number missing
ABA_ROUTING_NUMBER_NOT_ALLOWED	104	ABA routing number submitted for credit card transaction

String	*Code*	*Description*
ADPAY_PAYMENT_DESIGNATOR_MISSING	105	Missing payment designator for check draft or direct debit transaction
ADPAY_PAYMENT_DESIGNATOR_NOT_ ALLOWED	106	Payment designator submitted with credit card transaction
AMOUNT_MISSING	109	Amount missing
AMOUNT_NOT_ALLOWED	110	Amount not allowed
EXPIRATION_DATE_MISSING	113	Expiration date missing for credit card transactions
EXPIRATION_DATE_NOT_ALLOWED	114	Expiration date submitted for check draft or direct debit transactions
AUTHORIZATION_CODE_MISSING	115	Authorization code missing for post authorization
AUTHORIZATION_CODE_NOT_ALLOWED	116	Authorization code submitted for transaction other than post authorization
REFERENCE_NUMBER_MISSING	125	Required reference number is missing for a void transaction
CONFIGURATION_ERROR	131	Configuration error
INVALID_USER	132	Invalid user: The username passed by logon was not in the EdgCapture database.
INVALID_PASSWORD	133	Invalid password: The password passed by logon did not match the username.
ALREADY_ACKNOWLEDGED	134	Already acknowledged
ALREADY_LOGGED_ON	136	Client already logged on
NETWORK_TIMEOUT	141	Network timeout
WRONG_QUEUE	261	Wrong queue for a void transaction: voids can only be performed on transactions that are not in the captured queue.
NOT_IN_AUTHORIZED_QUEUE	291	The transaction is not in the authorize queue and can not be MarkedForCapture
AMOUNT_CHANGED	292	The original amount for this transaction has changed
FRAUD_SECURITY_RESPONSE_MATCH	302	Fraud security code match
FRAUD_SECURITY_RESPONSE_NO_MATCH	303	Fraud security code does not match code on card
FRAUD_SECURITY_RESPONSE_NOT_PROC ESSED	304	Fraud security code not processed by credit network
FRAUD_SECURITY_RESPONSE_NOT_ON_C ARD	305	Fraud security number is not available on card
FRAUD_SECURITY_RESPONSE_SHOULD_H AVE	306	Fraud security number is required for this transaction

String	*Code*	*Description*
FRAUD_SECURITY_RESPONSE_UNSUPPORTED	307	Fraud security response is not supported by the card
FRAUD_SECURITY_RESPONSE_INVALID	308	Credit network returned invalid fraud security resonse
FRAUD_SECURITY_RESPONSE_NO_DATA	309	Fraud security resonse invalid
ECCO_OVERTAX	310	Tax amount is more than total amount of transaction
ECCO_DECLINED_FSV	311	Transaction declined due to fraud security no match
ECCO_INVALID_FSV_FORMAT	312	Fraud security contains an invalid format
ECCO_CARD_SWIPE_NOT_LICENSED	322	Merchant not licensed for the card swipe option
AVS_EXACT_MATCH	323	Address verification match, zip and address
AVS_NO_MATCH	324	Address verification no match, zip and address
AVS_ADDRESS_MATCH	325	Address verification match, address only
AVS_NINE_ZIP_MATCH	326	Address verification match, zip only - 9 digits
AVS_FIVE_ZIP_MATCH	327	Address verification match, zip only - 5 digits
AVS_ADDRESS_UNAVAILABLE	328	Address verification, address unavailable
AVS_NOT_REQUIRED	329	Address verification, not tested or not required
AVS_NO_PARTICIPATION	330	Address verification, no issuer participation
AVS_NOT_SUPPORTED	331	Address verification, not supported
AVS_SYSTEM_UNAVAILABLE	332	Address verification, system unavailable
AVS_UNKNOWN	333	Address verification, unknown
ECCO_INVALID_ORDER_TYPE	334	Incorrect order type
AVS_MATCH	335	Address verification match, CM name and zip
AVS_MATCH_CMZIPADDR	336	Address verification match, CM name, zip and address
AVS_MATCH_CMADDR	337	Address verification match, CM name and address
AVS_MATCH_CM	338	Address verification match CM name
AVS_CMBAD_ZIPMATCH	339	Address verification match zip, CM name incorrect
AVS_CMBAD_ADDRMATCH	340	Address verification match address, CM name incorrect
AVS_CMBAD_ZIPADDRMATCH	341	Address verification match zip and address, CM N\name incorrect
AVS_CMZIPADDRBAD	342	Address verification, no match CM name, address and zip
INVALID_TOKEN	347	Specified token does not exisit in the database
INVALID_CDM_REQUEST	348	Unrecognized CDM request
CARDHOLDER_DATA_ALREADY_EXISTS	352	When creating a token, the cardholder data already exists in the database

String	Code	Description
TOKEN_NOT_ALLOWED	353	Token field was filled in but not expected
IO_EXCEPTION	1001	I/O Exception while parsing reply from ECCO Server.
XERCES_MISSING	1002	Xerces jar is missing. Parsing reply from ECCO Server is disabled
BAD_XML	1003	XML is bad. Parsing reply from ECCO Server is disabled.
UNKNOWN_REPLY	1004	Received unknown reply from the EdgCapture. Message status is unknown.
STATUS_MISSING	1005	ECCOStatus is missing from the reply. Message status is unknown.
NEVER_DID_LOGON	1006	ECCOClient never called logon.
ECCO_EXCEPTION	1007	EccoException during connection or messaging to EdgCapture
SAX_EXCEPTION	1008	Sax exception while parsing reply from EdgCapture.
MISSING_PAYMENT	1009	All payment information is missing.
TIMEOUT	1010	Processing of request has exceeded timeout period
FILE_NOT_FOUND_EXCEPTION	1011	Missing file
TRANSACTION_NOT_FOUND	1012	Specified transaction does not exist in the database.
MISSING_ECCO_PROPERTIES	1013	Failure to load ECCO.properties file.
ECCO_JNI_EXCEPTION	1014	JNI exception. JVM is not running or cannot be accessed.
MISSING_SSL_CERTIFY	1015	Certification failed or the application did not call certify method before logon.
INVALID_SALESTAX	1016	Sales tax amount not properly formatted or greater than amount
NO_RESULTS	1017	Received end-of-stream from socket

Appendix C: Monetary Transaction Data

The MonetaryTransactionData object's fields correspond to transaction data in the EdgCapture database. They can be used to submit transactions to EdgCapture for processing or to retrieve available transaction information from Edgcapture.

Field	***Size***	***Description***
ABANumber	9	American Bankers Association Routing number that identifies the bank issuing a check. REQUIRED if PaymentType is PAY_CHECK_DRAFT or PAY_DIRECT_DEBIT.
AccountNumber	48	Credit card or checking account number. REQUIRED for all monetary transactions.
AddressLine1	100	First line of customer's address
AddressLine2	100	Second line of customer's address
AddressVerification	1	Address verification results returned from the credit network. Values are: X - Address and 9-digit zip code match Y - Address and 5-digit zip code match A - Only address matches W - Only 9-digit zip code matches Z - Only 5-digit zip code matches N - No match U - No information available S - Service not available E - Not tested or required G - No issuer participation R - System not available 0 - Unknown
Amount	15	Transaction amount in format *nnnnnn.nn* with all decimal places expressed. Amount is always positive. Maximum size is 12,2. REQUIRED for all monetary transactions.
AuthorizationCode	10	Authorization code returned from the credit network for successfully authorized transaction
AuthorizationDate	10	Date the transaction was authorized in YYYY-MM-DD format returned from EdgCapture
CaptureDate	10	Date the transaction was captured (sent for settlement) in YYYY-MM-DD format returned from EdgCapture
City	50	Customer's city

Field	***Size***	***Description***
ECI	1	Electronic commerce indicator, which indicates how the customer's account data is transmitted to the application. Note that 6 is the usual value for web applications. Values are: 0 - Unknown 1 - Mail order/telephone order, single non-recurring 2 - Mail order/telephone order,recurring (for example, monthly subscription) 3 - Mail order/telephone order, installment 4 - Secure electronic transaction (SET) protocol using account holder certificate (not currently supported) 5 - SET using merchant certificate only (not currently supported) 6 - Non-SET, channel-encrypted security such as SSL (for example, a web site using SSL) 7 - Non-SET, non-channel-encrypted security (for example, unsecured e-mail)
ExpirationMonth	2	Credit card expiration month in format *nn.* REQUIRED for credit card payments.
ExpirationYear	2	Credit card expiration year in format *nn.* REQUIRED for credit card payments.
FirstName	60	Customer's first name
FraudSecurityResponse	30	Fraud security results returned from the payment network after checking submitted fraud security value against card issuer's record. Values are: 302 - Match of submitted and recorded value 303 - No match 304 - Value not processed by network or issuer 305 - Value not on card 306 - Value should be on card 307 - Unsupported by issuer 308 - Invalid data 309 - No response 0 - Not applicable to transaction
FraudSecurityValue	4	3- or 4-digit code separate from account number on credit cards; used to verify card-not-present transactions. Note that this value is passed in an authorization request, but it is NOT stored in the database for later retrieval in accordance with card issuer regulations.
LastName	60	Customer's last name
MerchantId	2	EdgCapture identifier for merchant account. REQUIRED for all transactions
MiddleInitial	1	Customer's middle initial
Notes	80	Additional information associated with the transaction
OEPId	2	EdgCapture identifier for the order entry process or subaccount for a given merchant. REQUIRED for all transactions.

Field	***Size***	***Description***
PaymentDesignator	30	The type of checking account: consumer or corporate. Values are: CHECK_DESIGNATOR_CONSUMER, CHECK_DESIGNATOR_CORPORATE. REQUIRED for payment type PAY_CHECK_DRAFT or PAY_DIRECT_DEBIT.
PaymentType	30	For a credit card account, the type of credit card; for a checking account, the type of account. Values for credit cards include: PAY_CREDIT_CARD, PAY_AMERICAN_EXPRESS, PAY_CARTE_BLANCHE, PAY_DINERS_CLUB, PAY_DISCOVER, PAY_MASTER_CARD, PAY_VISA. For checking: PAY_CHECK_DRAFT, PAY_DIRECT_DEBIT.
StateOrProvince	2	Two-character state or province code for customer address
Telephone	16	Customer's telephone number
QueueDateTime	20	Queuedatetime from EdgCapture in YYYY-MM-DD HH-MM:SS format. Queuedatetime is updated each time the transaction changes queue.
QueueName	30	Current queue of transaction in the EdgCapture database. Values include: QUEUE_AUTHORIZED, QUEUE_CAPTURED, QUEUE_CREDIT_PROBLEM, QUEUE_DECLINED, QUEUE_ERROR, QUEUE_HELD, QUEUE_INPUT, QUEUE_MARKED_FOR_CAPTURE, QUEUE_UNKNOWN_DISPOSITION, QUEUE_VOIDED
ReferenceNumber	15	Reference number in processing batch returned by the credit network or EdgCapture
TransactionId	30	Transaction identifier, which must be unique within a single merchant and OEP
TransactionStatusCode	30	Status code of the transaction from the EdgCapture database
TransactionType	30	Type of transaction. Possible values include: TRAN_AUTHORIZE, TRAN_CAPTURE_HELD_TRANSACTION, TRAN_CREDIT, TRAN_DECLINE_HELD_TRANSACTION, TRAN_VOID.
UserData1-10	40	Additional transaction information
ZipCode	12	Customer's zip or postal code

Appendix D: Editing Properties Files

The ECCO API uses two properties files to specify operating parameters: ECCO.properties and SimpleSocketServerEcco.properties. These files should be edited to correspond to your operating environment. Note that these files are case sensitive.

ECCO.properties

The ECCO.properties file specifies operating parameters for the java API that control the connection to EdgCapture and logging from the API.

Connection information

The file contains the following connection parameters that should be set to correspond to your operating environment. See “Appendix A: Test data” on page 187 for information about using a “Fake” connection for testing.

The Edgil delivery will include the keystore file necessary for secure connection to the EdgCapture server. The path to the keystore file, edgilca.keystore, is specified in ECCO.properties under the parameter com.edgil.ecco.sslKeyFile. Confirm the presence and location of edgilca.keystore and edit the parameter so the path is correct for your system.

Parameter	***Description***
com.edgil.ecco.serverConnectionType	Note that this parameter is case-sensitive; it must be entered as documented here with an initial upper-case character: Fake - Do not connect to EdgCapture. Ssl- Connect to EdgCapture.
com.edgil.ecco.serverIpAddress	IP address or hostname of the EdgCapture server
com.edgil.ecco.serverPortNum	Port number for the EccoServer task running on the EdgCapture server. Must match the value in the EdgCapture EIF database. Default is 2052.
com.edgil.ecco.connectionTimeout	Number of seconds until the connection times out when there is no activity. Default is 180.
com.edgil.ecco.sslKeyFile	Path to local keystore file.
com.edgil.ecco.logonTimeout	Something

Here is sample section of an ECCO.properties file that connects to EdgCapture:

```
com.edgil.ecco.serverConnectionType=Ssl
com.edgil.ecco.serverIpAddress=123.456.1.789
com.edgil.ecco.serverPortNum=2052
com.edgil.ecco.connectionTimeout=180
com.edgil.ecco.sslKeyFile=\\edgil\\java\\edgilca.keystore
com.edgil.ecco.logonTimeout=5
```

Connecting to multiple EdgCapture Servers

In order to connect to multiple Edgcapture servers from one environment, the API supports loading mutliple copies of the ecco.properties file.

Logging information

The sections controlling logging should be uncommented or commented out, depending upon the type of logging you want to use. Following are descriptions of the logging sections. Note that ConversionPattern in each appenders section specifies the content of a log entry: date, [thread], priority, class name, user-specified identifier, message.

No Logging

Uncomment this section to report only fatal errors to the console. Comment it out for any other logging situation. If this section is uncommented, you must also uncomment the stdout appenders section.

```
# No Logging
log4j.rootCategory=fatal, stdout
```

Logging both to the console and to a file

Uncomment this section to log to both the console and a file. If this section is uncommented, you must also uncomment both the stdout appenders and SocketApp appenders. Note that “info” logs all error levels.

```
# Logging both to console and to Socket Handler for file
# log4j.rootCategory=info, stdout, SocketApp
```

Logging to a file

Uncomment this section to log to a file only. If this section is uncommented, you must also uncomment the SocketApp appenders. Note that “info” logs all error levels.

```
#Logging to Socket Handler for file
log4j.rootCategory=info, SocketApp
```

Stdout appenders

Uncomment this section for no logging or logging to the console.

```
# Uncomment stdout appenders for no logging (fatal rootCategory)
# and for logging to stdout.
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
# Pattern with full date
log4j.appender.stdout.layout.ConversionPattern=%d [%t] %-5p %x
```

```
   %c - %m%n
```

SocketApp appenders

Uncomment this section for logging to a file. RemoteHost should be set to the host name of the local machine.

```
# Uncomment SocketApp appenders for logging to file.
# Leave them commented out for no logging.
# SimpleSocketServer needs to be running or errors will be
# written to the console
log4j.appender.SocketApp=org.apache.log4j.net.SocketAppender
log4j.appender.SocketApp.Port=3421
log4j.appender.SocketApp.RemoteHost=raven
#Pattern with full date
log4j.appender.SocketApp.layout.ConversionPattern=%d [%t] %-5p %x
   %c - %m%n
```

SimpleSocketServerEcco.properties

The SimpleSocketServerEcco.properties file specifies parameters for the log file. The default file applies to the java or C++ delivery on a Windows operating system. Users of other operating systems must edit the file to specify an appropriate pathname for the log file.

The default file specifies logging INFO and ERROR level messages to a file that is truncated and saved daily.

```
log4j.rootCategory=INFO, stdout,DailyRollingFile
```

This section applies to logging to the console. ConversionPattern specifies the content of a log entry: date, [thread], priority, class name, user-specified identifier, a dash, message text.

```
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d [%t] %-5p %c %x
   - %m%n
```

This section applies to the log file. Note that the directory path specified in File must exist on the logging system, or logging will fail. DatePattern specifies the format of the date extension appended to the previous day's file.

```
log4j.appender.DailyRollingFile=org.apache.log4j.
   DailyRollingFileAppender
log4j.appender.DailyRollingFile.File=c:\\edgil\\data\\ecco\\ecco.log
log4j.appender.DailyRollingFile.DatePattern='.'yyyy-MM-dd
log4j.appender.DailyRollingFile.layout=org.apache.log4j.
   PatternLayout
log4j.appender.DailyRollingFile.layout.ConversionPattern=%d [%t]
   %-5p %c %x - %m%n
```

Appendix E: Cardholder Data Management

Cardholder Data Management (CDM) is a feature of the ECCO API that manages sensitive cardholder data for clients so that they do not have to store this data on their system. ECCO will accept cardholder information and return a token to the client representing that information. CDM allows clients to execute recurring transactions for an account by storing a token for that account and sending that token to ECCO when processing a transaction.

CDM provides methods to allow the client to retrieve lists of expiring accounts and to update information for that account (except the account number) by referring to the account using the token.

CDM also supports the creation of a token without having to execute a transaction. This is convenient in the case that the client has acquired cardholder information for transactions that do not have to be processed immediately, i.e. scheduled transactions. The client sends in the cardholder data and ECCO returns the related token, which can then be used when a transaction needs to be processed against that account.

Terminology

Cardholder Data is data that refers to either a credit card, checking, or savings account. This includes the account holder's name and address, account number, and expiration date in addition to name, address, zip, and other cardholder data fields. Cardholder data may also refer to checking account data.

A Token is a string of characters that can be used in place of cardholder data when processing transactions through ECCO, or performing other cardholder data related functions.

CDM is Cardholder Data Management, the set of functionality described in this document.

Features

Backward Compatible
The use of CDM is optional. All future versions of the ECCO API will be completely backward compatible with existing client integrations.

Implicit Token Creation
The EdgCapture server will create a token for the client upon processing a transaction which was sent with cardholder data. This token is returned with the results of the transaction.

Explicit Token Creation
Clients can send account data to EdgCapture and receive a token before processing any transactions for that account. This prevents the client from having to store the account data for an unknown length of time before sending in the first transaction for that account. A new ECCO API request will support this functionality.

This request is called CreateToken. This request will also include level III data validation for the new account data.

If the cardholder data passed in for the CreateToken request is found in the database, the existing token is returned and the reply status will indicate that the cardholder data already exists.

Token Based Requests
Any request to the ECCO server that requires cardholder data to be filled in can be processed by substituting a token previously returned from a transaction or a create token request.

GetCardholderData
This request accepts a token as input and returns the associated cardholder data. The account number is masked according to the standard EdgCapture masking algorithm.

Update Cardholder Data
Cardholder data can be updated. Any field is allowed to be updated except for the account number. The server will maintain previous versions of the cardholder data for the account number, but the token will reference the cardholder data as of the most recent update. This request will also include level III data validation for the new account data.

Cardholder Data Uniqueness
A Token represents a unique set of cardholder data. If the client wants to change any account data, they must submit a new set of account data and get a new token. The one current exception is the extension of the expiration date described above.

Token Based Transaction Selection
The standard transaction selection parameters will be expanded to include a token value. All transactions using the account number that the token references will be returned, since the cardholder data may have been updated. For a given token the server will return all versions of cardholder data with the same account number, since from the caller's perspective it is the same account.

Security
A token can be used only by a valid logged on EdgCapture user, so the security restrictions applied to a user for regular transactions apply when using tokens. Tokens can be used to retrieve cardholder data, but the returned account number is masked according to the EdgCapture masking algorithm for credit card and checking account numbers. If a transaction is requested using a token, no cardholder data is returned in the reply. The addition of CDM will be implemented in such a way as to keep EdgCapture PABP compliant.

ECCO Class changes

Using Tokens in Monetary Transactions
MonetaryTransactionData now contains a token member variable. The token can be used in place of cardholder data for transactions. All current clients can ignore the token field if desired. The CDM changes are completely backward compatible.

The main purpose of using tokens is to avoid storing sensitive account information on your system. Once you receive a token from EdgCapture you can replace the cardholder data on your system with that token and use that token instead of the cardholder data for future transactions. This can be accomplished in two ways. Implicit token creation can be used if you need to process a transaction for a set of cardholder data. Explicit token creation allows you to create a token immediately without having to process a transaction until later.

Implicit token creation
Execute a monetary transaction with cardholder data specified. Upon return, get the token from the return data and store on your system (deleting cardholder data). Use that token for future transactions for that cardholder.

Explicit token creation
Use the new requestCreateToken call to get a token representing the cardholder data. Get the token from the results and store on your system (deleting cardholder data) use that token for future transactions for that cardholder.

Note: If you are using a token for a monetary transaction, make sure the account number field is not set. The server will use the account number and other cardholder data if the account number field is set, otherwise it will attempt to use the token.

New ECCO Client Requests
There are three new requests supporting CDM in the ECCOClient object. They are described briefly below. See the main part of this document for more details.

`requestCreateToken`

This request will create a set of cardholder data in the EdgCapture database. The cardholder data can be referenced using the returned token.

Upon sending the request, the cardholder data is populated with valid cardholder data (see definition of the cardholder data object). The token field is empty. Upon return, the token field is populated with the new token. If the passed cardholder data already exists, the existing token that references that data is returned.

If the AVSValidation parameter is true, the cardholder data is validated with respect to Address Verification processing. This is used for accessing beneficial processing rates for non consumer cards. For level III processing zip code and name is generally required.

`requestGetCardholderData`

This request will retrieve cardholder data from EdgCapture based on the token populated in the CardholderData parameter.

The token field of the CardholderData parameter should be filled in with the token that references the cardholder data being requested. If the cardholder data is found, the CardholderData are filled in with the retrieved information upon return.

`requestUpdateCardholderData`

This request will update an existing set of cardholder data in the EdgCapture database.

The CardholderData object should be filled in with a token that references an existing set of cardholder data, and all fields needing to be changed should be filled in with the new values. Account number can not be changed. If account number is filled in, error is returned.

After using the token to find the cardholder data, the server will update the cardholder data with all of the fields that are non-empty in the Cardholder data. Fields that are empty in the incoming data are ignored and those fields retain the same value as before the update.

Validation is performed on the data before update. If validation fails, an error is returned and the cardholder data is unchanged.

If the AVSValidation parameter is true, the cardholder data is validated with respect to Address Verification processing. This is used for accessing beneficial processing rates for non consumer cards. For Address Verification processing a zip code and name is required.