Monetra®

Java API Developer Reference

Monetra Java Reference v5.2
Updated November 2005

Table of Contents

1 Revision History

Date	*Rev.*	*Notes*
11/09/05	v5.2	Initial re-layout.

This page intentionally left blank.

2 Overview

2.1 Related Documentation

You will be required to review the Monetra Client Interface Protocol Specification (http://www.monetra.com/documentation.html) to cross-reference each transaction type, which will have multiple corresponding key/value pairs (ie. username, password, action, etc).

2.2 Introduction to the Monetra Java API

All applications using the Monetra JAVA JNI API must `import` the `com.mainstreetsoftworks.MONETRA` package that is installed as part of this API. The class name is `MONETRA` as well. Please refer to the API reference and examples below for usage of the APIs.

The Monetra (MCVE) Java API, which depends on libmonetra (C API), is designed to take advantage of all three of our "supported" communication methods, which include Drop-File, Unencrypted IP and Encrypted IP (SSL v3/TLS v1.0). Each method has its advantages and will be explained briefly below. Libmonetra is also the basis of the Perl, PHP and Java JNI modules, so the usage of those API's is nearly identical to Libmonetra itself, minus language semantics. In addition, this API was designed to be fully thread-safe and allows interleaving of transactions (streaming of transactions with out-of-order responses).

The Drop-File communication method is the most simplistic form of communication with Monetra. A transaction directory is specified where **.trn** (transaction) files are written, "picked up" and **.rtn** (response) files are written in reply. Advantages are the debug-ability and the fact that it does not require an IP stack to be present on the local machine. Although this method is not designed for networking, it is possible to share the transaction directory via NFS or SAMBA (for windows) to integrate with legacy applications. Due to security concerns, this should not be utilized for new integrations.

The unencrypted IP method is the fastest method of communication with Monetra. It requires the least amount of overhead and bypasses disk access. This method is perfect for locally "trusted" switched LANs or WANs, but should never be used on untrusted networks such as the Internet.

The encrypted IP (SSL) method is the most secure, requiring certificate verification and encryption to pass all data between the client and host. Most of the time, this is overkill for a local LAN or trusted WAN. SSL is most suitable for communication over the Internet or any untrusted network where the potential for eavesdropping is high. Newer Monetra releases also support client certificate validation which is available in this API.

For any feature/anomaly, requests or support questions regarding libmonetra, feel free to contact our support staff via e-mail at support@mainstreetsoftworks.com .

2.3 Obtaining and Installing Libmonetra and the Java API

Libmonetra may be obtained in source form from http://www.monetra.com/downloads.html or via ftp at ftp://ftp.monetra.com/pub/libmonetra. For 32bit Windows, it may also be obtained in binary form via a self-installing package from the same locations.

Please note: The Java API for Monetra (MCVE) depends on Libmonetra 5.x and it must be installed **before** attempting to compile/install the Java module.

The Java module is distributed only in source form, so it must be compiled. You may download the Java module source from the same location you obtained libmonetra. Once extracted, you will need to manually modify the included `Makefile` to properly set these variables:

`SSL_PREFIX=`	Installation location for OpenSSL
`JAVA_PREFIX=`	Base directory where java was installed
`MCFLAGS=`	Compiler C Flags needed to compile a shared object
`JNIEXT=`	Shared object extension used on OS (e.g. .so, .dll, .dylib, etc)
`MLDFLAGS=`	Linker Flags needed to create shared object
`JARDIR=`	Installation path for JAR files
`CLASSDIR=`	Installation path for CLASS files
`LIBDIR=`	Installation path for Java JNI Shared Objects

You may notice that we have pre-defined some of these parameters for some OS's such as Linux, Solaris, FreeBSD and MacOSX. It's possible that simply uncommenting the predefined parameters for your OS will allow you to compile the module.

Some other parameters that might need to be tweaked include:

`LDADD=`	Libraries that need to be linked into the shared object
`JNINAME=`	The name of the output JNI library
`CC=`	If the compiler is not cc or gcc, this flag must be set
`OUTNAME=`	How to set the output filename for the linker

Once those flags are set, you may compile and install the JNI library by using these commands:

```
      make
      make install
```

You can test to make sure you have successfully compiled the library by running the built-in test:

```
      make test         # basic sale transaction test
      make test2        # basic report test
      make test3        # first test from examples in this document
      make test4        # second test from examples in this
                        # documentation
      make guitest      # test a GUI Java application to Monetra
```

2.4 Caveats to the Monetra JAVA JNI API

Since the Monetra JAVA API is based off of JNI (Java Native Interface), this means that there is underlying C code which is not cross-platform, meaning it does not generate machine-independent byte-code which may be distributed to any machine. If you are a software developer, and plan to distribute an application which utilizes the Monetra JAVA JNI API, this means that you must compile the Monetra Java JNI API for each architecture and OS you plan to support, and include all the necessary files which include the binary, non-java objects. If this is an issue, we'd recommend utilizing the XML protocol via HTTP(S) POST, or integrate via the direct IP/SSL socket support within Monetra.

3 Using This Guide

LibMonetra only performs simple connection and transaction management facilities to the Monetra engine. Its API was created to be as minimalistic and simple to use as possible. It will pass the transaction set (a set of key/value pairs) to the Monetra engine and return to you a response. It provides additional parsing facilities for dealing with comma delimited responses as well. Please reference the Monetra Client Interface Protocol Specification located at http://www.monetra.com/documentation.html for the expected key/value pairs for each transaction and responses to those requests.

The basics for performing transactions for this guide include initializing the library, establishing a connection, structuring and sending one or more transactions, reading results and closing the connection/de-initializing the library.

You will note in this API that all parameters to functions are prefixed with an [in] or [out] tag which indicates if you are receiving data into that parameter.

[in] means this is an input parameter used to tell the function what it needs to perform. This parameter should be passed normally (e.g. not by reference).

[out] means that the parameter's memory address will be updated upon return. You need to make sure this variable is passed by reference.

*Please reference the examples in this document for basic API usage.

4 Constants and Data Types

4.1 Constants for ReturnStatus

4.1.1 M_ERROR

Value:	`-1`
Description:	Critical error. Status unknown

4.1.2 M_FAIL

Value:	`0`
Description:	Transaction or Audit Failed

4.1.3 M_SUCCESS

Value:	`1`
Description:	Transaction or Audit succeeded

4.2 Constants for CheckStatus

4.2.1 M_DONE

Value:	`2`
Description:	Transaction is complete

4.2.2 M_ERROR

Value:	`-1`
Description:	An error has occurred. Status unknown.

4.2.3 M_PENDING

Value:	`1`
Description:	Still waiting on transaction response from Monetra

5 Java Functions

5.1 Initialization/Destruction of Library

5.1.1 MONETRA

Prototype: `public MONETRA(String cafile)`

Description: Class constructor for MONETRA
Call as

```
      private static MONETRA conn1 = new MONETRA(null);
```

or similar to initialize the class.

Return Value: Class

Parameter Descriptions:

[in] cafile: SSL CA (Certificate Authority) file for verification of remote SSL servers (DEPRECATED, pass null, use M_SetSSL_CAfile instead).

5.1.2 close

Prototype: `public int close()`

Description: Used to disconnect and free memory associated with the class, if needed before automatic garbage collection takes place.

Return Value: 1 on success, 0 on failure

Parameter Descriptions:

N/A

5.2 Initialization of Connections and Management

5.2.1 Connect

Prototype:	`public synchronized int Connect()`
Description:	once all connection parameters have been set, this function establishes the connection to the Monetra daemon
Return Value:	1 on success, 0 on failure

Parameter Descriptions:
N/A

5.2.2 ConnectionError

Prototype:	`public String ConnectionError()`
Description:	if M_Connect returns a failure, this function may provide some text as an insight into what went wrong (such as timeout, or connection refused)
Return Value:	textual error message associated with connection

Parameter Descriptions:
N/A

5.2.3 MaxConnTimeout

Prototype:	`public void MaxConnTimeout(int maxtime)`
Description:	sets how long libmonetra should try to connect to the Monetra server. This only has an effect when there are network problems and sets the socket into non blocking mode. Only relevant for IP or SSL connections.
Return Value:	1 on success, 0 on failure

Parameter Descriptions:

[in] `maxtime:`	maximum amount of time in seconds to wait to establish IP/SSL connection

5.2.4 SetBlocking

Prototype:	`public synchronized int SetBlocking(int tf)`
Description:	specifies whether to wait for a transaction to finish before returning from a M_TransSend or (legacy) M_Sale etc. (blocking), or to return immediately and make client check status (non-blocking)
Return Value:	1 on success, 0 on failure

Parameter Descriptions:

[in] `tf:`	1 if blocking is desired, 0 if blocking is not desired

5.2.5 SetDropFile

Prototype: `public int SetDropFile(String df_location)`
Description: sets the M_CONN parameter to use the Drop-File method of communication
Return Value: 1 on success, 0 on failure

Parameter Descriptions:

[in] `df_location:`	directory to write transaction files

5.2.6 SetIP

Prototype: `public int SetIP(String host, int port)`
Description: sets the M_CONN parameter to use the IP method of communication
Return Value: 1 on success, 0 on failure

Parameter Descriptions:

[in] `host:`	hostname or ip address to establish IP connection
[in] `port:`	port associated with ip/hostname

5.2.7 SetSSL

Prototype: `public int SetSSL(String host, int port)`
Description: sets the M_CONN parameter to use the SSL method of communication
Return Value: 1 on success, 0 on failure

Parameter Descriptions:

[in] `host:`	hostname or ip address to establish SSL connection
[in] `port:`	port associated with ip/hostname

5.2.8 SetSSL_CAfile

Prototype: `public int SetSSL_CAfile(String path)`
Description: sets the CA file location before establishing a connection to a running Monetra engine. Used to verify the remote host's SSL certificate.
Return Value: 1 on success, 0 on failure

Parameter Descriptions:

[in] `path:`	CA (Certificate Authority) file path

5.2.9 SetSSL_Files

Prototype: `public int SetSSL_Files(String sslkeyfile, String sslcertfile)`
Description: sets the client certificate and key used for verification if the remote Monetra engine has client SSL certificate verification enabled
Return Value: 1 on success, 0 on failure

Parameter Descriptions:

[in] `sslkeyfile:`	path of key file for client certificate
[in] `sslcertfile:`	path of certificate file for client certificate

5.2.10 SetTimeout

Prototype: `public synchronized int SetTimeout(int secs)`
Description: sets the maximum amount of time a transaction can take before timing out. This values gets sent to the Monetra engine, the engine sends the TIMEOUT response to libmonetra, libmonetra never times out (and SHOULD NOT)
Return Value: 1 on success, 0 on failure

Parameter Descriptions:

[in] `timeout:`	maximum duration in seconds to wait for completion of transaction

5.2.11 ValidateIdentifier

Prototype: `public int ValidateIdentifier(int tf)`
Description: tells the API whether or not the identifiers used for transactions should be validated from within the connection structure before assuming they are correct. Since the transaction identifier is actually a pointer address, passing an incorrect address can cause segmentation faults without verification. This is usually not necessary for C programs, but is helpful for writing modules for other languages.
Return Value: 1 on success, 0 on failure

Parameter Descriptions:

[in] `tf:`	1 if verification of transaction identifiers is desired, 0 if not desired [default 0]

5.2.12 VerifyConnection

Prototype: `public void VerifyConnection(int tf)`
Description: tells Monetra whether or not to send a PING request to the Monetra server once a connection has been established. Default is TRUE. This will probably only be used if trying to connect to a Monetra version < 2.1.
Return Value: none

Parameter Descriptions:

[in] `tf:`	1 if SSL server certification verification is desired, 0 if not [default 0]

5.2.13 VerifySSLCert

Prototype: `public void VerifySSLCert(int tf)`

Description: tells Monetra whether or not to verify that the SSL certificate provided by Monetra has been signed by a proper CA. Obviously, this is only applicable if using SSL connectivity.

Return Value: none

Parameter Descriptions:

[in] `tf`:	1 if SSL server certification verification is desired, 0 if not [default 0]

5.3 Sending Transactions to Monetra

5.3.1 CheckStatus

Prototype: `public synchronized int CheckStatus(long identifier)`
Description: returns the state of the transaction, whether or not processing has been complete or is still pending
Return Value: M_PENDING (1) if still being processed, M_DONE (2) if complete, <= 0 on critical failure

Parameter Descriptions:
[in] `identifier:` reference for transaction as returned by M_TransNew()

5.3.2 CompleteAuthorizations

Prototype: `public long [] CompleteAuthorizations()`
Description: gets how many transactions have been completed and loads the list of identifiers into listings
Return Value: number of transactions in the current connection queue which are complete (fully processed)

Parameter Descriptions:
[out] `listings:` returns an identifier for each listing which is complete. Should free () the array returned here. Must not be NULL.

5.3.3 DeleteTrans

Prototype: `public synchronized void DeleteTrans(long identifier)`
Description: removes a transaction from the queue that was initialized with M_TransNew
Return Value: none

Parameter Descriptions:
[in] `identifier:` reference for transaction as returned by M_TransNew()

5.3.4 Monitor

Prototype: `public synchronized int Monitor()`
Description: Performs all communication with the Monetra server. If this function never gets called, no transactions will be processed. Function is non-blocking, meaning it will return immediately if there is nothing to be done.
Return Value: 1 on success (connection alive), 0 on disconnect, -1 on critical error

Parameter Descriptions:
N/A

5.3.5 TransInQueue

Prototype:	`public synchronized int TransInQueue()`
Description:	returns the total number of transactions in the queue, no matter what state they are in or if they have been sent or not.
Return Value:	number of transactions in the current connection queue

Parameter Descriptions:
N/A

5.3.6 TransKeyVal

Prototype:	`public int TransKeyVal(long identifier, String key, String value)`
Description:	adds a key/value pair for a transaction to be sent to Monetra
Return Value:	1 on success, 0 on failure

Parameter Descriptions:

[in] `identifier:`	reference for transaction as returned by M_TransNew()
[in] `key:`	key as referenced in the Monetra Client Interface Specification
[in] `value:`	value as referenced for key in Monetra Client Interface Specification

5.3.7 TransNew

Prototype:	`public synchronized long TransNew()`
Description:	starts a new transaction. This must be called to obtain an identifier before any Transaction Parameters may be added.
Return Value:	reference for transaction

Parameter Descriptions:
N/A

5.3.8 TransactionsSent

Prototype:	`public int TransactionsSent()`
Description:	checks to make sure the SEND queue for IP and SSL connections is empty. Useful for determining connection problems to Monetra.
Return Value:	number of transactions sent to Monetra from this connection (that have not already been deleted).

Parameter Descriptions:
N/A

5.3.9 TransSend

Prototype: `public synchronized int TransSend(long identifier)`
Description: finalizes a transaction and sends it to the Monetra server
Return Value: 1 on success, 0 on failure

Parameter Descriptions:
[in] `identifier:` reference for transaction as returned by M_TransNew()

5.4 Dealing with Responses from Monetra

5.4.1 GetCell

Prototype: `public String GetCell(long identifier, String column, int row)`
Description: gets a single cell from comma-delimited data (position independent M_ParseCommaDelimited must be called first.
Return Value: data for particular cell

Parameter Descriptions:

[in] `identifier:`	reference for transaction as returned by M_TransNew()
[in] `column:`	text key (header) name for cell
[in] `row:`	row number

5.4.2 GetCellByNum

Prototype: `public String GetCellByNum(long identifier, int column, int row)`
Description: gets a single cell from comma-delimited data (position dependent M_ParseCommaDelimited must be called first.
Return Value: data for particular cell

Parameter Descriptions:

[in] `identifier:`	reference for transaction as returned by M_TransNew()
[in] `column:`	integer value for column number
[in] `row:`	row number

5.4.3 GetCommaDelimited

Prototype: `public String GetCommaDelimited(long identifier)`
Description: gets the raw comma-delimited data
Return Value: raw transaction data returned by Monetra

Parameter Descriptions:

[in] `identifier:`	reference for transaction as returned by M_TransNew()

5.4.4 GetHeader

Prototype: `public String GetHeader(long identifier, int column_num)`
Description: retrieval of a header by column number from comma-delimited data. M_ParseCommaDelimited must be called first.
Return Value: text name for column header

Parameter Descriptions:

[in] `identifier:`	reference for transaction as returned by M_TransNew()
[in] `column_num:`	column number to retrieve header name

5.4.5 IsCommaDelimited

Prototype: `public int IsCommaDelimited(long identifier)`
Description: a quick check to see if the response that has been returned is comma-delimited or a standard response
Return Value: 1 if response is comma-delimited, 0 if not

Parameter Descriptions:
[in] `identifier:` reference for transaction as returned by M_TransNew()

5.4.6 NumColumns

Prototype: `public int NumColumns(long identifier)`
Description: the number of columns in a comma-delimited responseM_ParseCommaDelimited must be called first.
Return Value: number of columns for comma-delimited data

Parameter Descriptions:
[in] `identifier:` reference for transaction as returned by M_TransNew()

5.4.7 NumRows

Prototype: `public int NumRows(long identifier)`
Description: the number of rows in a comma-delimited response. M_ParseCommaDelimited must be called first.
Return Value: number of rows for comma delimited data

Parameter Descriptions:
[in] `identifier:` reference for transaction as returned by M_TransNew()

5.4.8 ParseCommaDelimited

Prototype: `public synchronized int ParseCommaDelimited(long identifier)`
Description: tells libmonetra to use its internal parsing commands to parse the comma delimited response. This MUST be called before calls to M_GetCell, M_GetCellByNum, M_GetHeader, M_NumRows, and M_NumColumns.
Note: If you call M_ParseCommaDelimited, you can no longer call MonetraGetCommaDelimited because ParseCommaDelimited destroys the data.
Return Value: 1 on success, 0 on failure

Parameter Descriptions:
[in] `identifier:` reference for transaction as returned by M_TransNew()

5.4.9 ResponseKeys

Prototype: `public String [] ResponseKeys(long identifier)`
Description: retrieves the response keys (parameters) returned from the Monetra engine for the particular transaction. Useful so you can pull the value using M_ResponseParam() for each key.
Return Value: array of strings which are the available keys in the response

Parameter Descriptions:
[in] `identifier:` reference for transaction as returned by M_TransNew()

5.4.10 ResponseParam

Prototype: `public String ResponseParam(long identifier, String key)`
Description: This function is used to retrieve the response key/value pairs from the Monetra Engine, as specified in the Monetra Client Interface Protocol Specification.
Return Value: value associated with the key requested. NULL if not found.

Parameter Descriptions:
[in] `identifier:` reference for transaction as returned by M_TransNew()
[in] `key:` Response Parameter key as defined in the Monetra Client Interface Specification

5.4.11 ReturnStatus

Prototype: `public int ReturnStatus(long identifier)`
Description: returns a success/fail response for every transaction. If a detailed code is needed, please see M_ReturnCode
Return Value: 1 if transaction successful (authorization), 0 if transaction failed (denial)

Parameter Descriptions:
[in] `identifier:` reference for transaction as returned by M_TransNew()

5.5 Miscellaneous Functions

5.5.1 SSLCert_gen_hash

Prototype:	`public String SSLCert_gen_hash(String filename)`
Description:	generates hash content of client certificate for adding restrictions to user connections
Return Value:	certificate hash, or NULL on error

Parameter Descriptions:

[in] `filename:`	path to client certificate file

5.5.2 uwait

Prototype:	`public int uwait(int length)`
Description:	a microsecond sleep timer that uses select() with a NULL set to obtain a more efficient, cross-platform, thread-safe usleep();
Return Value:	1 on success, 0 on failure

Parameter Descriptions:

[in] `length:`	time in micro seconds to delay (1/1000000)

6 Examples

6.1 Full Transaction Examples

6.1.1 Basic Sale Transaction Code

```
import com.mainstreetsoftworks.MONETRA;

class TestApp {
      private static String MYHOST=   "localhost";
      private static int MYPORT=      8444;
      private static String MYUSER=   "test-user";
      private static String MYPASS=   "test-pass";
      private static String MYMETHOD= "SSL";
      private static String MYCAFILE= "/usr/local/monetra/CAfile.pem";
      private static boolean MYVERIFYSSL=     true;

      private static MONETRA conn1 = null;
      private static long identifier = 0;
      private static int i = 0;
      private static java.lang.String response_keys[];

      public static void main(String[] args) {
            /* Initialize Class */
            conn1 = new MONETRA(null);
            if (MYMETHOD.compareToIgnoreCase("SSL") == 0) {
                  /* Set up SSL Connection Location */
                  if (conn1.SetSSL(MYHOST, MYPORT) == 0) {
                        System.out.println(
                              "Could not set method to SSL");
                        /* Free memory associated with conn */
                        conn1.close();
                        return;
                  }
                  /* Set up information required to verify
                   * certificates */
                  if (MYVERIFYSSL) {
                        if (conn1.SetSSL_CAfile(MYCAFILE) == 0) {
                              System.out.println(
                                    "Could not set SSL CAFile.  " +
                                    "Does the file exist?");
                              conn1.close();
                              return;
                        }
                        conn1.VerifySSLCert(1);
                  }
            } else if (MYMETHOD.compareToIgnoreCase("IP")
                  == 0) {
```

```
                /* Set up IP Connection Location */
                if (conn1.SetIP(MYHOST, MYPORT) == 0) {
                System.out.println(
                      "Could not set method to IP");
                      /* Free memory associated with conn */
                      conn1.close();
                      return;
                }
          } else {
                System.out.println(
                      "Invalid method '" + MYMETHOD +
                      "' specified!");
                /* Free memory associated with conn */
                conn1.close();
                return;
          }

          /* Set to non-blocking mode, means we must do
           * a M_Monitor() loop waiting on responses
           * Please see next example for blocking-mode */
          if (conn1.SetBlocking(0) == 0) {
                System.out.println(
                      "Could not set non-blocking mode");
                /* Free memory associated with conn */
                conn1.close();
                return;
          }
          /* Set a timeout to be appended to each transaction
          * sent to Monetra */
          if (conn1.SetTimeout(30) == 0) {
                System.out.println("Could not set timeout");
                /* Free memory associated with conn */
                conn1.close();
                return;
          }

          /* Connect to Monetra */
          if (conn1.Connect() == 0) {
                System.out.println(
                      "Connection failed: " +
                      conn1.ConnectionError());
                /* Free memory associated with conn */
                conn1.close();
                return;
          }

          /* Allocate new transaction */
          identifier=conn1.TransNew();

          /* User credentials */
          conn1.TransKeyVal(identifier, "username", MYUSER);
          conn1.TransKeyVal(identifier, "password", MYPASS);
          /* Transaction Type */
```

```
            conn1.TransKeyVal(identifier, "action", "sale");
            /* Transaction Data */
            conn1.TransKeyVal(identifier, "account",
                  "4012888888881");
            conn1.TransKeyVal(identifier, "expdate", "0512");
            conn1.TransKeyVal(identifier, "amount", "12.00");
            conn1.TransKeyVal(identifier, "ptrannum", "99999");

            /* Add transaction to outgoing buffer */
            if (conn1.TransSend(identifier) == 0) {
                  System.out.println(
                        "Transaction improperly structured, possibly"
                        + "not enough info");
                  /* Free memory associated with conn */
                  conn1.close();
                  return;
            }

            /* Communication loop with Monetra. Loop until
             * transaction is complete */
            while (conn1.CheckStatus(identifier) == conn1.M_PENDING) {
                  if (conn1.Monitor() != 1) {
                        /* Disconnect has occurred, or other
                         * critical error */
                        System.out.println(
                              "Unexpected disconnect: " +
                              conn1.ConnectionError());
                  }
                  conn1.uwait(20000); /* Microsecond sleep */
            }

            /* Check success or Fail */
            if (conn1.ReturnStatus(identifier) == conn1.M_SUCCESS) {
                  System.out.println("Transaction successful!");
            } else if (conn1.ReturnStatus(identifier) ==
                  conn1.M_FAIL) {
                  System.out.println("Transaction failed!");
            }

            /* Get results */
            response_keys = conn1.ResponseKeys(identifier);
            System.out.println("Response Keys: Values");
            for (i=0; i<response_keys.length; i++) {
                  System.out.println(response_keys[i] + " : " +
                        conn1.ResponseParam(identifier,
                              response_keys[i]));
            }
            /* Optionally clean up transaction memory, this is
             * automatically free()'d when the connection is
             * destroyed */
            conn1.DeleteTrans(identifier);

            /* Clean up connection, and library instance :
                 * OPTIONAL*/
               conn1.close();
        }
}
```

6.1.2 Requesting and Interpreting Reports

```
import com.mainstreetsoftworks.MONETRA;

class TestApp2 {
      private static String MYHOST=   "localhost";
      private static int MYPORT=      8444;
      private static String MYUSER=   "test-user";
      private static String MYPASS=   "test-pass";
      private static String MYMETHOD= "SSL";
      private static String MYCAFILE= "/usr/local/monetra/CAfile.pem";
      private static boolean MYVERIFYSSL=     true;

      private static MONETRA conn1 = null;
      private static long identifier = 0;
      private static int i = 0;
      private static int j = 0;
      private static int rows = 0;
      private static int columns = 0;

      public static void main(String[] args) {
            /* Initialize Class */
            conn1 = new MONETRA(null);

            if (MYMETHOD.compareToIgnoreCase("SSL") == 0) {
                  /* Set up SSL Connection Location */
                  if (conn1.SetSSL(MYHOST, MYPORT) == 0) {
                        System.out.println(
                              "Could not set method to SSL");
                        /* Free memory associated with conn */
                        conn1.close();
                        return;
                  }
                  /* Set up information required to verify
                   * certificates */
                  if (MYVERIFYSSL) {
                        if (conn1.SetSSL_CAfile(MYCAFILE) == 0) {
                              System.out.println(
                                    "Could not set SSL CAFile.  " +
                                    "Does the file exist?");
                              conn1.close();
                              return;
                        }
                        conn1.VerifySSLCert(1);
                  }
            } else if (MYMETHOD.compareToIgnoreCase("IP") == 0) {
                  /* Set up IP Connection Location */
                  if (conn1.SetIP(MYHOST, MYPORT) == 0) {
                        System.out.println(
                              "Could not set method to IP");
                        /* Free memory associated with conn */
                        conn1.close();
                        return;
                  }
            } else {
                  System.out.println("Invalid method '" + MYMETHOD +
                        "' specified!");
                  /* Free memory associated with conn */
                  conn1.close();
                  return;
            }

            /* Set to blocking mode, means we do not have to
             * do a M_Monitor() loop, M_TransSend() will do this
             * for us */
            if (conn1.SetBlocking(1) == 0) {
                  System.out.println(
                        "Could not set non-blocking mode");
                  /* Free memory associated with conn */
                  conn1.close();
                  return;
            }

            /* Set a timeout to be appended to each transaction
             * sent to Monetra */
            if (conn1.SetTimeout(30) == 0) {
                  System.out.println("Could not set timeout");
                  /* Free memory associated with conn */
                  conn1.close();
                  return;
            }

            /* Connect to Monetra */
            if (conn1.Connect() == 0) {
                  System.out.println("Connection failed: " +
                        conn1.ConnectionError());
                  /* Free memory associated with conn */
                  conn1.close();
                  return;
            }

            /* Allocate new transaction */
            identifier=conn1.TransNew();

            /* User credentials */
            conn1.TransKeyVal(identifier, "username", MYUSER);
            conn1.TransKeyVal(identifier, "password", MYPASS);
            /* Transaction Type */
            conn1.TransKeyVal(identifier, "action", "admin");
            /* Transaction Data */
            conn1.TransKeyVal(identifier, "admin", "GUT");
            /* Additional Auditing parameters may be specified
             * Please consult the Monetra Client Interface Protocol */

            if (conn1.TransSend(identifier) == 0) {
```

```
            System.out.println("Communication Error: " +
                    conn1.ConnectionError());
            /* Free memory associated with conn */
            conn1.close();
            return;
        }

        /* We do not have to perform the Monitor() loop
         * because we are in blocking mode */
        if (conn1.ReturnStatus(identifier) != conn1.M_SUCCESS) {
            System.out.println("Audit failed");
            conn1.close();
            return;
        }

        if (conn1.IsCommaDelimited(identifier) == 0) {
            System.out.println(
                    "Not a comma delimited response!");
            conn1.close();
            return;
        }

        /* Print the raw, unparsed data */
        System.out.println ("Raw Data:\r\n" +
            conn1.GetCommaDelimited(identifier));

        /* Tell the API to parse the Data */
        if (conn1.ParseCommaDelimited(identifier) == 0) {
            System.out.println(
                    "Parsing comma delimited data failed");
            conn1.close();
            return;
        }

        /* Retrieve each number of rows/columns */
        rows=conn1.NumRows(identifier);
        columns=conn1.NumColumns(identifier);

        /* Print all the headers separated by |'s */
        for (i=0; i<columns; i++) {
            if (i != 0) System.out.print("|");
            System.out.print(conn1.GetHeader(identifier, i));
        }
        System.out.print("\r\n");

        /* Print one row per line, each cell separated by |'s */
        for (j=0; j<rows; j++) {
            for (i=0; i<columns; i++) {
                if (i != 0) System.out.print("|");
                System.out.print(
                        conn1.GetCellByNum(identifier, i, j));
            }
            System.out.print("\r\n");
            }

            /* Use M_GetCell instead of M_GetCellByNum if you need a
             * specific column, as the results will allow for
             * position-independent searching of the results. The
             * ordering of returned headers may be different between
             * Monetra versions, so that is _highly_ recommended */

            /* Optionally free transaction, though M_DestroyConn()
             * will do this for us */
            conn1.DeleteTrans(identifier);

            /* Clean up and close */
            conn1.close();
      }
}
```