

Messaging Services for Microsoft Windows

MailEnable API Guide

Issue Date: 30th June 2026

Table of Contents

1 Introduction 3
2 MailEnable Constructs 4
2.1 Address Formatting 4
2.2 Message Command Files 5
3 Developing Connectors 8
3.1 Implementing a Connector 8
3.2 Considerations 9
4 Developing MTA Pickup Events 10
4.1 Developing a Pickup Event Application 10
4.2 Developing a Pickup Event Class 14
4.2.1 Developing the Object 14
4.2.2 Integrating the COM Pickup Event 15
5 Developing Mailbox Delivery Events 17
6 More Information 18

1 Introduction

MailEnable exposes an array of interfaces that you can use to automate its behaviour or extend the application. This API Guide explains these interfaces and provides examples of how to configure MailEnable programmatically.

IMPORTANT: You are not permitted to distribute MailEnable components in any product or extension. The target environment for your application or extension must either have the Standard. Professional or Enterprise Edition of MailEnable installed.

MailEnable takes reasonable effort to ensure that the information provided in this API guide is accurate at the time of publishing. From time to time, the API may change as extensions emerge and such changes will be communicated through revisions to the API guide.

---

2 MailEnable Constructs

This section outlines some basic construction and underlying principals with respect to the inner workings of MailEnable.

2.1 Address Formatting

An internal MailEnable address is made of two core parts. Firstly, there is the Connector Descriptor and secondly there is the addressing detail. The exact syntax is shown below:

Syntax:

[Connector Acronym: Connector Address Details]

Examples:

MailEnable Internal Address	Explanation
[SF:POSTOFFICE/MAILBOX]	The location of mailbox (MAILBOX) on postoffice (POSTOFFICE) using the SF Connector (Postoffice Connector)
[SMTP:User@domain]	The SMTP Address of a user at the prescribed domain using the SMTP Connector
[LS:POSTOFFICE/LISTNAME]	The location of list mailbox (LISTNAME) on postoffice (POSTOFFICE) using the LS Connector (List Connector).

When a mail connector receives mail, it resolves the addressed recipients to an internal address format. Some of these recipients will be local, and others will be relayed to non-local users. The connector will produce a command file containing all resolved recipients and a message file containing the actual data. This information is stored in the Connectors Spooling directory.

2.2 Message Command Files

For most messages queues there are two files per message. The first is called a **command file** and the second is the message body. The command file contains details concerning the delivery of the message (such as sender, retry count, etc.). These details will vary depending on the queue the message is in.

The command file for the SMTP connector is located in the Mail Enable\Queues\SMTP\Outgoing directory. It has the file extension .MAI. The first part of the name should be a GUID to avoid overwriting existing files. The command file corresponds to the actual message that is located in the Mail Enable\Queues\SMTP\Outgoing\Messages directory. The message file must always be written before you create the command file, in order to avoid services reading the command file and not finding the message.

To determine the location of the queues, you would read the directory path from the Windows registry key:

[HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Mail Enable\Mail Enable\Connectors]
"Connector Root Directory"="C:\\Program Files (x86)\\Mail Enable\\QUEUES"

Under this path connectors will have their own queues.

The command file has the following format:

```
DomainName=[Setting]<CRLF>
CommandType=[Setting]<CRLF>
Server=[Setting]<CRLF>
Recipients=[Setting]<CRLF>
Sender=[Setting]<CRLF>
Retries=[Setting]<CRLF>
NextSendTime=[Setting]<CRLF>
TimeAcquired=[Setting]<CRLF>
MessageID=[Setting]<CRLF>
Priority=[Setting]<CRLF>
Status=[Setting]
ClientIP=[Setting]<CRLF>
Account=[Setting]<CRLF>
AuthenticationStatus=[Setting]<CRLF>
Subject=[Setting]
```

The values of the items are described below:

DomainName:	This is the name of the domain that will be connected to. If this is a new message, you can leave this blank (i.e. just have DomainName=), as MailEnable will resolve it. This is only used only when sending messages out the SMTP Connector and is written only by the connector once MX lookup has

	occurred.
CommandType:	Indicates whether the message is a non-delivery receipt. For new messages this should be "Normal", and for system generated messages it is “NDR”. When NDR messages are sent outbound from the SMTP connector they are sent with a NULL sender address (<>).
Server:	The name of the Windows server that is to process the file. You do not need to set this at this stage. You do not need to set this value unless you wish to denote the message as being owned by a particular server (ie: if you are clustering).
Recipients:	The email address of the recipients. You can one or more recipients. If you have more than one recipient, separate the addresses with a semi-colon, such as: `[SMTP:email@domain.com];[SMTP:anotheremail@domain.com]` Keep the total recipient length to be less than 24000 bytes.
Sender:	The email address of the sender. This is formatted using the MailEnable address format.
Retries:	The amount of send retries the message has had.
NextSendTime:	The message will be tried to be sent if the current time is greater than this time. This is measured in the number of seconds since midnight January 1, 1970, in local time. To send a message immediately, set this to the same value as TimeAcquired.
TimeAquired:	The time the message was created. This is used when checking to see if the message has expired. This is measured in the number of seconds since midnight January 1, 1970, in local time.
MessageID:	The name of the message file. Both the message and command files should have the same filename.
Priority:	Unused. Usually set to "Normal".
Status:	What status the message is in. Set this to "Unsent" for new messages. When a message is being sent, this becomes "Sending".
ClientIP:	This is the IP address of the originating client if available.
Account:	This field denotes the 'Owner’ postoffice of message. This records the postoffice who would get billed for the message (by an integrated billing system).
AuthenticationStatus:	This field indicates whether the sender of the message has authenticated before inserting the message into the MailEnable Queues. In the case of SMTP Inbound messages, this entry is populated with a value other than zero under the following circumstances:

	• The sender authenticated via SMTP Authentication, sets value to "1" • The sender has been permitted to relay from an authorized IP address range – sets value to "2". Note: the authentication flag value of "1" will take precedence over the value of "2". I.e.: If the sender has authenticated and are authorized to relay, the value will be set to "1".
Subject	The subject of the message.

Example command file:

```
DomainName=mailenable.com
CommandType=NORMAL
Server=
Recipients=[SMTP:info@mailenable.com]
Sender=[SMTP:support@mailenable.com]
Retries=2
NextSendTime=1009515032
TimeAcquired=1009407032
MessageID=D9880414C29A4DEC94C02457718EE.MAI
Priority=Normal
Status=Unsent
Server=MESRV01
```

3 Developing Connectors

MailEnable allows you to define custom connectors to facilitate mail connectivity to backend systems. Such connectors are useful to integrate the likes of telephony/communications, printing and external systems with MailEnable.

For example, it may be desirable to publish access to a backend order entry system as an SMTP address to allow orders to be distributed to a backend system. Orders would be received by SMTP and placed in a backend Order Management System. Backend system could also place messages into its own inbound queue for distribution/notification, etc.

3.1 Implementing a Connector

To implement your own connector, you must first give the connector a name.

In our example, we will use the MEOES as the connector name (for Order Entry System).

STEP 1: Creating the Connector Queues

Firstly, you must create the following Queue directories:

C:\Program Files\Mail Enable\Queues\MEOES\Inbound\Messages
C:\Program Files\Mail Enable\Queues\MEOES\Outbound\Messages

Note: These assume the default location for the Queue directory. It is also important to make sure that the entire paths outlined above exist. Specifically, the Messages Subdirectories are imperative.

STEP 2: Registering the Connector

You then need to create the following registry key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Mail Enable\Mail Enable\Connectors\MEOES
```

STEP 3: Allocating Mailboxes Connector Addresses

Now that the connector queues are created you are able to add address maps to that connector in the ADDRESS-MAP.TAB file. This is exactly how the SMTP, POPC, PostOffice and List Connectors work.

An Example address map follows:

Message arrives via SMTP -> MTA determines connector mapping -> MEOES connector -> System

This is represented in the address map file as:

Source Address = "[SMTP:OrderSystemPayment@domain.com]"
Target Address = "[MEOES:PAYMENT]”

This rule would mean that any messages arriving for the SMTP Address would be placed in the Queue with a command file addressing to [MEOES:PAYMENT], with a corresponding message in the Messages directory. The reverse can also be achieved.

STEP 4: Creating the Connector

You now need to write the connector itself. This is simply a program that reads to and from the MEOES queues you just created.

A good starting point for writing your own connector is the sample MTA pickup event provided on the MailEnble Web Site. The MailEnable System Manual also explains how connectors work.

3.2 Considerations

When developing custom connectors you should try to avoid using generic connector names - ie: connector names that are logically intended to be part of MailEnable. These would specifically include FAX, PRN, SMS and TEL queue names. Any custom queue names should be prefixed with at least a two vendor allocated letters to distinguish it from other queues.

This will prevent any custom connector conflicting with address maps of any integrated connectors, as well as reducing the likelihood of cross vendor conflicts.

4 Developing MTA Pickup Events

When the MTA moves a message between connectors, an optional executable file or COM DLL can be invoked. This is called a pickup event. The MTA pickup event will pass the mail message filename to the external application/COM DLL.

For example, if you wrote a VB script the adds some text to the end of each email you could activate this application through the pickup event. The application/DLL receives the messagefilename and connectortype as parameters.

e.g.: Program MessageFileName ConnectorType

Where:

Program is the program filename,
MessageFileName is the name of the message file
ConnectorType is the type of messages (ie. SMTP, LS, SF).

Be aware that the directory path to the message is not passed to the program. You will need to read the directory path from the registry in the external application.

4.1 Developing a Pickup Event Application

A Pickup Event Application is a Windows Executable that is shelled as a message passes through the MTA. As mentioned earlier, the shelled executable accepts a space delimited set of parameters.

Since the parameters do not contain the physical queue path, we first need to access the registry and determine the location of the connectors queue and construct the location of the message command file and the message itself.

Simple Example:

```
using System;
using System.IO;
using Microsoft.Win32;
using System.Diagnostics;

namespace PluginTest
{
    class Program
    {
        private static bool DebugMode = false;
        private static string LogFilePath = "";

        static int Main(string[] args)
        {
            string messageId = "";
            string connectorCode = "";

            try
            {
```

```
            if (args.Length < 2)
            {
                return 1;
            }

            messageId = args[0];
            connectorCode = args[1];

            DebugMode = HasDebugSwitch(args);

            if (DebugMode)
            {
                LogFilePath = BuildDebugLogFilePath(messageId, connectorCode);
                Log("Starting plugin test");
                Log("Message ID: " + messageId);
                Log("Connector Code: " + connectorCode);
            }

            string dataDirectory = GetMailEnableDataDirectory();

            if (String.IsNullOrEmpty(dataDirectory))
            {
                Log("Could not read MailEnable Data Directory from registry.");
                return 2;
            }

            string queueRoot = Path.Combine(dataDirectory, "QUEUES");

            string msgCommandFile = Path.Combine(
                Path.Combine(
                    Path.Combine(queueRoot, connectorCode),
                    "Inbound"),
                messageId);

            string msgFile = Path.Combine(
                Path.Combine(
                    Path.Combine(
                        Path.Combine(queueRoot, connectorCode),
                        "Inbound"),
                    "Messages"),
                messageId);

            Log("Data Directory: " + dataDirectory);
            Log("Command File: " + msgCommandFile);
            Log("Message File: " + msgFile);

            if (!File.Exists(msgFile))
            {
                Log("Message file does not exist.");
                return 3;
            }

            bool performAction = MessageContainsExecutableReference(msgFile);

            if (performAction)
            {
                Log("Matched .exe. Deleting message files.");

                DeleteFileIfExists(msgCommandFile);
                DeleteFileIfExists(msgFile);

                Log("Message deleted.");
            }
            else
            {
                Log("No action required.");
            }

            Log("Finished plugin test.");
            return 0;
        }
```

```
            catch (Exception ex)
            {
                Log(
                    "Could not process pickup event for Connector: " +
                    connectorCode +
                    " Message ID: " +
                    messageId +
                    " Error: " +
                    ex.ToString());

                return 10;
            }
        }

        private static bool MessageContainsExecutableReference(string msgFile)
        {
            using (StreamReader reader = new StreamReader(msgFile))
            {
                string line;

                while ((line = reader.ReadLine()) != null)
                {
                    if (line.IndexOf(".exe", StringComparison.OrdinalIgnoreCase) >= 0)
                    {
                        Log("Found .exe in message file.");
                        return true;
                    }
                }
            }

            return false;
        }

        private static string GetMailEnableDataDirectory()
        {
            const string valueName = "Data Directory";

            string[] possibleKeys = new string[]
            {
                @"SOFTWARE\Mail Enable\Mail Enable",
                @"SOFTWARE\Wow6432Node\Mail Enable\Mail Enable"
            };

            for (int i = 0; i < possibleKeys.Length; i++)
            {
                string keyPath = possibleKeys[i];

                Log("Checking registry key: HKLM\\" + keyPath);

                using (RegistryKey key = Registry.LocalMachine.OpenSubKey(keyPath))
                {
                    if (key == null)
                    {
                        Log("Registry key not found.");
                        continue;
                    }

                    object value = key.GetValue(valueName);

                    if (value == null)
                    {
                        Log("Registry value not found: " + valueName);
                        continue;
                    }

                    string dataDirectory = value.ToString();

                    if (!String.IsNullOrEmpty(dataDirectory))
                    {
                        Log("Found Data Directory: " + dataDirectory);
                        return dataDirectory;
```

```
                    }
                }
            }

            return "";
        }

        private static void DeleteFileIfExists(string filePath)
        {
            if (File.Exists(filePath))
            {
                File.Delete(filePath);
                Log("Deleted file: " + filePath);
            }
            else
            {
                Log("File did not exist, nothing to delete: " + filePath);
            }
        }

        private static bool HasDebugSwitch(string[] args)
        {
            for (int i = 0; i < args.Length; i++)
            {
                if (String.Compare(args[i], "/debug", true) == 0 ||
                    String.Compare(args[i], "-debug", true) == 0 ||
                    String.Compare(args[i], "--debug", true) == 0)
                {
                    return true;
                }
            }

            return false;
        }

        private static string BuildDebugLogFilePath(string messageId, string
connectorCode)
        {
            string safeMessageId = MakeSafeFileName(messageId);
            string safeConnectorCode = MakeSafeFileName(connectorCode);

            string logDir = Path.Combine(
                Path.GetTempPath(),
                "MailEnablePluginTest");

            string fileName =
                "PluginTest_" +
                safeConnectorCode +
                "_" +
                safeMessageId +
                "_" +
                Process.GetCurrentProcess().Id.ToString() +
                ".log";

            return Path.Combine(logDir, fileName);
        }

        private static string MakeSafeFileName(string value)
        {
            if (String.IsNullOrEmpty(value))
                return "empty";

            char[] invalidChars = Path.GetInvalidFileNameChars();

            for (int i = 0; i < invalidChars.Length; i++)
            {
                value = value.Replace(invalidChars[i], '_');
            }

            return value;
        }
```

```
        private static void Log(string message)
        {
            if (!DebugMode)
                return;

            try
            {
                string logDir = Path.GetDirectoryName(LogFilePath);

                if (!String.IsNullOrEmpty(logDir) && !Directory.Exists(logDir))
                {
                    Directory.CreateDirectory(logDir);
                }

                string line =
                    DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss.fff") +
                    " [" +
                    Process.GetCurrentProcess().Id.ToString() +
                    "] " +
                    message +
                    Environment.NewLine;

                File.AppendAllText(LogFilePath, line);
            }
            catch
            {
                // Never allow logging failure to affect message processing.
            }
        }
    }
}
```

4.2 Developing a Pickup Event Class

MailEnable's MTA also allows you to call COM components as well as shelled MTA pickup events. The advantage of calling a COM component is that it can run in the same context and process address space as the MTA process itself.

Using the COM Pickup Event also can have considerable disadvantages (but careful programming of the Pickup Event can overcome the issue).

A considerable disadvantage in calling the COM DLL is that any faults occurring in the COM object will be inherited by the MTA and therefore could make the MTA unstable. To avoid this, authors of such COM components should take care to manage exceptions/errors and ensuring that they do not cause an unknown state.

4.2.1 Developing the Object

The most practical/simplest tool for developing a COM Pickup Event is Visual Basic 6.0 You should create a new project as an ActiveX DLL and name the project/object and classes to be indicative to the purpose of the pickup event.
You should then add a function to one of the classes as a public Method called "Execute". The function should take a string as a parameter.

Example:

```
Public Function Execute (ByVal Params As String) As Long
        '
        'Params: Same parameters as those passed to a pickup event.
        '


        ‘
        ‘ Function should return 1 if Success, 0 if failure
        ‘
End Function
```

When a message passes through the MTA, the pickup event will call the execute method on the COM object, passing the same parameters as are passed to a shelled Pickup Event.

The simplest way to test the functionality is to run the MTA in debug mode on your development server and have the Execute function simply display a message box.

Eg:

```
Public Function Execute (ByVal Params As String) As Long
        '
        'Params: Same parameters as those passed to a pickup event.
        '
        MsgBox Params


        ‘
        ‘ Function should return 1 if Success, 0 if failure
        ‘
        Execute = 1
End Function
```

When developing the pickup event for production, it is important that the respective and Microsoft recommended practices for creating unattended library COM components are followed. Specifically, the component should not display and UI and should be compiled with the Unattended Execution option.

4.2.2 Integrating the COM Pickup Event

To configure the MTA to shell a COM Component, you need to enter the qualified object.class name into the following registry key

KeyRoot	HKLM\SOFTWARE\Mail Enable\Mail Enable\AGENTS\MTA
Value Name	Pickup Event Class Name
Value	String containing class name
Value Type	REG_SZ
Example Value	MyObject.MTAPE

You also need to configure the MTA to call the COM Pickup Event via an additional Registry Key:

KeyRoot	HKLM\SOFTWARE\Mail Enable\Mail Enable\AGENTS\MTA
Value Name	Pickup Event Class Enabled
Value	0 = Off, 1 = On
Value Type	REG_DWORD
Example Value	MyObject.MTAPE

5 Developing Mailbox Delivery Events

A Mailbox Delivery Event is optionally triggered when the Postoffice Connector attempts to deliver mail to a mailbox. The Postoffice Connector will check whether a delivery event is configured for the mailbox and will execute the specified executable.

When the delivery event is triggered, the following parameters are passed on the command line to the specified executable (they are delimited by a space):

PostofficeName Mailbox MessageID

Note: You should note that Mailbox Delivery Events generate different parameters to Mail Transfer Agent Pickup Events. You cannot therefore simply run code designed for a Pickup Event as a Delivery Event (although it should be quite simple to modify the parameter list expected by the target executable).

6 More Information

More Information on MailEnable Developer Resources can be found at:

https://www.mailenable.com/developers

You can also access the MailEnable developers forum at:

https://forum.mailenable.com/viewforum.php?f=4