Using Caché Internet Utilities

Version 2018.1
2024-04-03

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Using Caché Internet Utilities
Caché Version 2018.1 2024-04-03

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

List of Tables

About This Book

This manual helps programmers use some of the key classes in the %Net package, which provide an easy-to-use interface for a number of useful Internet protocols. Because the class documentation for this package is fairly extensive, this manual provides a quick, organizing overview, rather than an intensive look at every parameter, property, and method. It is assumed that you are familiar with the protocols and third-party tools mentioned in this manual.

This book consists of the following chapters:

- Sending HTTP Requests
- Sending and Receiving Email
- Creating, Writing, and Reading MIME Messages
- Using FTP
- Using SSH
- Sending and Receiving IBM WebSphere MQ Messages
- Other Caché %Net Tools

For a detailed outline, see the table of contents.

Also see the following books:

- *Creating Web Services and Web Clients in Caché* describes how to create Caché web services and web clients.
- *Securing Caché Web Services* describes how to add security elements to Caché web services and web clients.

For general information, see the *InterSystems Documentation Guide*.

1

Sending HTTP Requests

This chapter describes how to send HTTP requests (such as POST or GET) and process the responses. It includes the following topics:

1.1 Introduction to HTTP Requests

You create an instance of %Net.HttpRequest to send HTTP requests of various kinds and receive the responses. This object is equivalent to a web browser, and you can use it to make multiple requests. It automatically sends the correct cookies and sets the `Referer` header as appropriate.

To create an HTTP request, use the following general process:

1. Create an instance of %Net.HttpRequest.
2. Set properties of this instance to indicate the web server to communicate with. The basic properties are as follows:
   - Server specifies the IP address or machine name of the web server. The default is `localhost`.

     Note: Do not include `http://` or `https://` as part of the value of Server. This causes `ERROR #6059: Unable to open TCP/IP socket to server http:/`.

- Port specifies the port to connect to. The default is 80.

3. Optionally set other properties and call methods of your HTTP request, as described in "Specifying Other HTTP Request Properties."
4. Then send an HTTP request, by calling the **Get()** method or other methods of your instance of %Net.HttpRequest, as described in "Sending the HTTP Request."

   You can make multiple requests from your instance, which will automatically handle cookies and the `Referer` header.

   Note: If you created this HTTP request for use with the Ensemble outbound adapter (EnsLib.HTTP.OutboundAdapter), then instead use the methods of that adapter to send the request.

5. Use the same instance of %Net.HttpRequest to send additional HTTP requests if needed. By default, Caché keeps the TCP/IP socket open so that you can reuse the socket without closing and reopening it.

   For additional information, see "Managing Socket Reuse."

The following shows a simple example:

ObjectScript

```
set request=##class(%Net.HttpRequest).%New()
set request.Server="tools.ietf.org"
set request.Https=1
set request.SSLConfiguration="TEST"
set status=request.Get("/html/rfc7158")
```

For information on the Https and SSLConfiguration properties, see "Using SSL to Connect," later in this chapter. Also, for a more complete version of this example, see "Accessing the HTTP Response."

1.2 Providing Authentication

If the destination server requires login credentials, your HTTP request can include an HTTP `Authorization` header that provides the credentials. The following subsections provide the details:

- Authenticating a Request When Using HTTP 1.0
- Authenticating a Request When Using HTTP 1.1
- Specifying the Authorization Header Directly
- Enabling Logging for HTTP Authentication

If you are using a proxy server, you can also specify login credentials for the proxy server; to do so, set the ProxyAuthorization property; see "Using a Proxy Server." For details, see the class documentation for %Net.HttpRequest.

1.2.1 Authenticating a Request When Using HTTP 1.0

For HTTP 1.0, to authenticate an HTTP request, set the Username and Password properties of the instance of %Net.HttpRequest. The instance then creates the HTTP `Authorization` header based on that username and password, using the Basic Access Authentication (RFC 2617). Any subsequent request sent by this %Net.HttpRequest will include this header.

Important: Make sure to also use SSL (see "Using SSL to Connect"). In Basic authentication, the credentials are sent in base-64 encoded form and thus can be easily read.

1.2.2 Authenticating a Request When Using HTTP 1.1

For HTTP 1.1, to authenticate an HTTP request, in most cases, just set the Username and Password properties of the instance of %Net.HttpRequest. When an instance of %Net.HttpRequest receives a 401 HTTP status code and `WWW-Authenticate` header, it attempts to respond with an `Authorization` header that contains a supported authentication scheme. The first scheme that is supported and configured for Caché is used. By default, it considers these authentication schemes, in order:

1. Negotiate (SPNEGO and Kerberos, per RFC 4559 and RFC 4178)
2. NTLM (NT LAN Manager Authentication Protocol)
3. Basic (Basic Access Authentication as described in RFC 2617)

Important: If there is a chance that Basic authentication will be used, make sure to also use SSL (see "Using SSL to Connect"). In Basic authentication, the credentials are sent in base-64 encoded form and thus can be easily read.

On Windows, if the Username property is not specified, Caché can instead use the current login context. Specifically, if the server responds with a 401 status code and a `WWW-Authenticate` header for SPNEGO, Kerberos, or NTLM, then Caché uses the current operating system username and password to create the `Authorization` header.

The details are different from the HTTP 1.0 case, as follows:

- If authentication succeeds, Caché updates the CurrentAuthenticationScheme property of the %Net.HttpRequest instance to indicate the authentication scheme that it used for the most recent authentication.
- If an attempt to get an authentication handle or token for a scheme fails, Caché saves the underlying error to the AuthenticationErrors property of the %Net.HttpRequest instance. The value of this property is a **$LIST**, in which each item has the form *scheme* `ERROR:` *message*

Negotiate and NTLM are supported only for HTTP 1.1 because these schemes require multiple round trips, and HTTP 1.0 requires the connection to be closed after each request/response pair.

1.2.2.1 Variations

If you know the authentication scheme or schemes allowed by the server, you can bypass the initial round trip from the server by including an `Authorization` header that contains the initial token for the server for a chosen scheme. To do this, set the InitiateAuthentication property of the %Net.HttpRequest instance. For the value of this property, specify the name of a single authorization scheme allowed by the server. Use one of the following values (case-sensitive):

- `Negotiate`
- `NTLM`
- `Basic`

If you want to customize the authentication schemes to use (or change their order in which they are considered), set the AuthenticationSchemes of the %Net.HttpRequest instance. For the value of this property, specify a comma-separated list of authentication scheme names (using the exact values given in the previous list).

1.2.3 Specifying the Authorization Header Directly

For either HTTP 1.0 or HTTP 1.1 (if applicable to your scenario), you can specify the HTTP `Authorization` header directly. Specifically, you set the Authorization property equal to the authentication information required by the user agent for the resource that you are requesting.

If you specify the Authorization property, the Username and Password properties are ignored.

1.2.4 Enabling Logging for HTTP Authentication

To enable logging for the HTTP authentication, enter the following in the Terminal:

```
zn "%SYS"
set ^%ISCLOG=2
set ^%ISCLOG("Category","HttpRequest")=5
```

Log entries are written to the *^%ISCLOG("Data")* global node. To write the log to a file for easier readability, enter the following (still within the %SYS namespace):

```
do ##class(%OAuth2.Utils).DisplayLog("filename")
```

Where *filename* is the name of the file to create. The directory must already exist. If the file already exists, it is overwritten.

To stop logging, enter the following (still within the %SYS namespace):

```
set ^%ISCLOG=0
set ^%ISCLOG("Category","HttpRequest")=0
```

1.3 Specifying Other HTTP Request Properties

Before you send an HTTP request (see "Sending the HTTP Request"), you can specify its properties, as described in the following sections:

- The Location Property
- Specifying the Internet Media Type and Character Encoding
- Using a Proxy Server
- Using SSL to Connect
- The HTTPVersion, Timeout, WriteTimeout, and FollowRedirect Properties
- Specifying Default Values for HTTP Requests

You can specify default values for all properties of %Net.HttpRequest, as specified in the section listed last.

1.3.1 The Location Property

The Location property specifies the resource that you are requesting from the web server. If you set this property, then when you call the **Get()**, **Head()**, **Post()**, or **Put()** method, you can omit the location argument.

For example, suppose that you are sending an HTTP request to the URL `http://machine_name/cache/index.html`

In this case, you would use the following values:

Table 1–1: Example Properties for %Net.HttpRequest

Properties	Value
Server	`machine_name`
Location	`cache/index.html`

1.3.2 Specifying the Internet Media Type and Character Encoding

You can use the following properties to specify the Internet media type (also called *MIME type*) and character encoding in your instance of %Net.HttpRequest and its response:

- ContentType specifies the `Content-Type` header, which specifies the Internet media type of the request body. The default type is none.

  Possible values include `application/json`, `application/pdf`, `application/postscript`, `image/jpeg`, `image/png`, `multipart/form-data`, `text/html`, `text/plain`, `text/xml`, and many others. For an extensive list, see http://www.iana.org/assignments/media-types.

- The ContentCharset property controls the desired character set for any content of the request if the content is of type text (`text/html` or `text/xml` for example). If you do not specify this property, Caché uses the default encoding of the Caché server.

  Note: If you set this property, you must first set the ContentType property.

- The NoDefaultContentCharset property controls whether to include an explicit character set for content of type `text` if you have not set the ContentCharset property. By default, this property is false.

  If this property is true, then if you have content of type `text` and if you have not set the ContentCharset property, no character set is included in the content type; this means that the character set iso-8859–1 is used for the output of the message.

- The WriteRawMode property affects the entity body (if included). It controls how the body of the request is written. By default, this property is false and Caché writes the body in the encoding that is specified in the request headers. If this property is true, Caché writes the body in RAW mode (performing no translation of the character set).

- The ReadRawMode property controls how the body of the response is read. By default, this property is false and Caché assumes that the body is in the character set specified in the response headers. If this property is true, Caché reads the body in RAW mode (performing no translation of the character set).

1.3.3 Using a Proxy Server

You can send an HTTP request via a proxy server. In order to set this up, specify the following properties of your HTTP request:

- ProxyServer specifies the host name of the proxy server to use. If this property is not null, the HTTP request is directed to this machine.

- ProxyPort specifies the port to connect to, on the proxy server.

- ProxyAuthorization specifies the `Proxy-Authorization` header, which you must set if a user agent must authenticate itself with a proxy. For the value, use the authentication information required by the user agent for the resource that you are requesting. Also see "Providing Login Credentials."

- ProxyHTTPS controls whether the HTTP request is for an HTTPS page, rather than a normal HTTP page. This property is ignored if you have not specified a proxy server. This property changes the default port on the target system to 443, the proxy port. Also see "Using SSL to Connect."

- ProxyTunnel specifies whether to establish a tunnel through the proxy to the target HTTP server. If true, the request uses the HTTP CONNECT command to establish a tunnel. The address of the proxy server is taken from the ProxyServer and ProxyPort properties. If ProxyHttps is true, then once the tunnel is established, Caché negotiates the SSL connection. In this case, the Https property is ignored because the tunnel establishes a direct connection with the target system.

For details, see the class documentation for %Net.HttpRequest.

1.3.4 Using SSL to Connect

The %Net.HttpRequest class supports SSL connections. To send the request via SSL, do the following:

1. Set the SSLConfiguration property to the name of the activated SSL/TLS configuration to use.

   For information on creating and managing SSL/TLS configurations, see "Using SSL/TLS with Caché" in the *Caché Security Administration Guide*. The SSL/TLS configuration includes an option called **Configuration Name**, which is the string to use in this setting.

2. Also do one of the following, depending on whether you are using a proxy server:

   - If you are not using a proxy server, set the Https property to true.
   - If you are using a proxy server, set the ProxyHTTPS property to true.

     In this case, to use an SSL connection to the proxy server itself, set the Https property to true.

Note that when you use an SSL connection to a given server, the default port on that server is assumed to be 443 (the HTTPS port). For example, if you are not using a proxy server and Https is true, this changes the default Port property to 443.

Also see "Using a Proxy Server."

1.3.4.1 Server Identity Checking

By default, when an instance of %Net.HttpRequest connects to a SSL/TLS secured web server, it checks whether the certificate server name matches the DNS name used to connect to the server. If these names do not match, the connection is not permitted. This default behavior prevents "man in the middle" attacks and is described in RFC 2818, section 3.1; also see RFC 2595, section 2.4.

To disable this check, set the SSLCheckServerIdentity property to 0.

1.3.5 The HTTPVersion, Timeout, WriteTimeout, and FollowRedirect Properties

%Net.HttpRequest also provides the following properties:

- HTTPVersion specifies the HTTP version to use when requesting a page. The default is `"HTTP/1.1"`. You can also use the `"HTTP/1.0"`.
- Timeout specifies how long, in seconds, to wait for a response from the web server. The default is 30 seconds.
- WriteTimeout specifies how long, in seconds, to wait for a write the web server to complete. By default it will wait indefinitely. The minimum accepted value is 2 seconds.
- FollowRedirect specifies whether to automatically follow redirection requests from the web server (signalled by the HTTP status codes in the range 300–399). The default is true if you are using GET or HEAD; otherwise it is false.

1.3.6 Specifying Default Values for HTTP Requests

You can specify default values for all properties of %Net.HttpRequest.

- To specify a default value that apply to all namespaces, set the global node `^%SYS("HttpRequest","propname")` where *"propname"* is the name of the property.
- To specify a default value for one namespace, go to that namespace and set the node `^SYS("HttpRequest","propname")`

(The ^%SYS global affects the entire installation, and the ^SYS global affects the current namespace.)

For example to specify a default proxy server for all namespaces, set the global node
`^%SYS("HttpRequest","ProxyServer")`

1.4 Setting and Getting HTTP Headers

You can set values for and get values of the HTTP headers.

Each of the following properties of %Net.HttpRequest contains the value of the HTTP header that has the corresponding name. These properties are automatically calculated if you do not set them:

- Authorization
- ContentEncoding
- ContentLength (This property is read-only.)
- ContentType (Specifies the Internet media type (MIME type) of the `Content-Type` header.)
- ContentCharset (Specifies the `charset` part of the `Content-Type` header. If you set this, you must *first* set the ContentType property.)
- Date
- From
- IfModifiedSince
- Pragma
- ProxyAuthorization
- Referer
- UserAgent

The %Net.HttpRequest class provides general methods that you can use to set and get the main HTTP headers. These methods ignore `Content-Type` and other entity headers.

ReturnHeaders()

Returns a string containing the main HTTP headers in this request.

OutputHeaders()

Writes the main HTTP headers to the current device.

GetHeader()

Returns the current value for any main HTTP header that has been set in this request. This method takes one argument, the name of the header (not case-sensitive); this is a string such as `Host` or `Date`

SetHeader()

Sets the value of a header. Typically you use this to set nonstandard headers; most of the usual headers are set via properties such as Date. This method takes two arguments:

1. The name of the header (not case-sensitive), without the colon (`:`) delimiter; this is a string such as `Host` or `Date`

2. The value of that header

You cannot use this method to set entity headers or read-only headers (`Content-Length` and `Connection`).

For details, see the class documentation for %Net.HttpRequest.

1.5 Managing Keep-alive Behavior

If you reuse the same instance of %Net.HttpRequest to send multiple HTTP requests, by default, Caché keeps the TCP/IP socket open so that Caché does not need to close and reopen it.

If you do not want to reuse the TCP/IP socket, do either of the following:

- Set the SocketTimeout property to 0.
- Add the 'Connection: close' HTTP header in your HTTP request. To do so, add code like the following before you send the request:

  ObjectScript

  ```
  Set sc=http.SetHeader("Connection","close")
  ```

  Note that the HTTP request headers are cleared after each request, so you would need to include this code before each request.

The SocketTimeout property of %Net.HttpRequest specifies the window of time, in seconds, during which Caché will reuse a given socket. This timeout is intended to avoid using a socket that may have been silently closed by a firewall. The default value for this property is 115. You can set it to a different value.

1.6 Handling HTTP Request Parameters

When you send an HTTP request (see "Sending the HTTP Request"), you can include parameters in the *location* argument; for example: `"/test.html?PARAM=%25VALUE"` sets PARAM equal to %VALUE.

You can also use the following methods to control how your instance of %Net.HttpRequest handles parameters:

InsertParam()

Inserts a parameter into the request. This method takes two string arguments: the name of the parameter and the value of the parameter. For example:

ObjectScript

```
do req.InsertParam("arg1","1")
```

You can insert more than one value for a given parameter. If you do so, the values receive subscripts starting with 1. Within other methods, you use these subscripts to refer to the intended value.

DeleteParam()

Deletes a parameter from the request. The first argument is the name of the parameter. The second argument is the subscript for the value to delete; use this only if the request contains multiple values for the same parameter.

CountParam()

Counts the number of values that are associated with a given parameter.

GetParam()

Gets the value of a given parameter in the request. The first argument is the name of the parameter. The second argument is the default value to return if the request does not have a parameter with this name; the initial value for this default is the null value. The third argument is the subscript for the value to get; use this only if the request contains multiple values for the same parameter.

IsParamDefined()

Checks whether a given parameter is defined. This method returns true if the parameter has a value. The arguments are the same as for **DeleteParam()**.

NextParam()

Retrieves the name of the next parameter, if any, after sorting the parameter names via **$Order()**.

ReturnParams()

Returns the list of parameters in this request.

For details, see the class documentation for %Net.HttpRequest.

1.7 Including a Request Body

An HTTP request can include either a request body or form data. To include a request body, do the following:

1. Create an instance of %GlobalBinaryStream or a subclass. Use this instance for the EntityBody property of your HTTP request.
2. Use the standard stream interface to write data into this stream. For example:

ObjectScript

```
Do oref.EntityBody.Write("Data into stream")
```

For example, you could read a file and use that as the entity body of your custom HTTP request:

ObjectScript

```
set file=##class(%File).%New("G:\customer\catalog.xml")
set status=file.Open("RS")
if $$$ISERR(status) do $System.Status.DisplayError(status)
set hr=##class(%Net.HttpRequest).%New()
do hr.EntityBody.CopyFrom(file)
do file.Close()
```

1.7.1 Sending a Chunked Request

If you use HTTP 1.1, you can send an HTTP request in chunks. This involves setting the Transfer-Encoding to indicate that the message is chunked, and using a zero-sized chunk to indicate completion.

Chunked encoding is useful when the server is returning a large amount of data and the total size of the response is not known until the request is fully processed. In such a case, you would normally need to buffer the entire message until the content length could be computed (which %Net.HttpRequest does automatically).

To send a chunked request, do the following:

1. Create a subclass of %Net.ChunkedWriter, which is an abstract stream class that defines an interface for writing data in chunks.

   In this subclass, implement the **OutputStream()** method.

2. In your instance of %Net.HttpRequest, create an instance of your %Net.ChunkedWriter subclass and populate it with the request data that you want to send.

3. Set the EntityBody property of your %Net.HttpRequest instance equal to this instance of %Net.ChunkedWriter.

   When you send the HTTP request (see "Sending the HTTP Request"), it calls the **OutputStream()** method of the EntityBody property.

In your subclass of %Net.ChunkedWriter, the **OutputStream()** method should examine the stream data, decide whether to chunk it and how to do so, and invoke the inherited methods of the class to write the output.

The following methods are available:

WriteSingleChunk()

Accepts a string argument and writes the string as non-chunked output.

WriteFirstChunk()

Accepts a string argument. Writes the appropriate Transfer-Encoding heading to indicate a chunked message, and then writes the string as the first chunk.

WriteChunk()

Accepts a string argument and writes the string as a chunk.

WriteLastChunk()

Accepts a string argument and writes the string as a chunk, followed by a zero length chunk to mark the end.

If non-null, the TranslateTable property specifies the translation table to use to translate each string as it is written. All of the preceding methods check this property.

1.8 Sending Form Data

An HTTP request can include either a request body or form data. To include a form data, use the following methods:

InsertFormData()

Inserts form data into the request. This method takes two string arguments: the name of the form item and the associated value. You can insert more than one value for a given form item. If you do so, the values receive subscripts starting with 1. Within other methods, you use these subscripts to refer to the intended value

DeleteFormData()

Deletes form data from the request. The first argument is the name of the form item. The second argument is the subscript for the value to delete; use this only if the request contains multiple values for the same form item.

CountFormData()

Counts the number of values associated with a given name, in the request.

IsFormDataDefined()

Checks whether a given name is defined

NextFormData()

Retrieves the name of the next form item, if any, after sorting the names via **$Order()**.

For details on these methods, see the class documentation for %Net.HttpRequest.

1.8.1 Example 1

After inserting the form data, you generally call the **Post()** method. For example:

ObjectScript

```
Do httprequest.InsertFormData("element","value")
Do httprequest.Post("/cgi-bin/script.CGI")
```

1.8.2 Example 2

For another example:

ObjectScript

```
Set httprequest=##class(%Net.HttpRequest).%New()
set httprequest.SSLConfiguration="MySSLConfiguration"
set httprequest.Https=1
set httprequest.Server="myserver.com"
set httprequest.Port=443
Do httprequest.InsertFormData("portalid","2000000")
set tSc = httprequest.Post("/url-path/")
Quit httprequest.HttpResponse
```

1.9 Inserting, Listing, and Deleting Cookies

%Net.HttpRequest automatically manages cookies sent from the server; if the server sends a cookie, your instance of %Net.HttpRequest will return this cookie on the next request. (For this mechanism to work, you need to reuse the same instance of %Net.HttpRequest.)

Use the following methods to manage cookies within your instance of %Net.HttpRequest:

InsertCookie()

Inserts a cookie into the request. Specify the following arguments:

1. Name of the cookie.
2. Value of the cookie.
3. Path where the cookie should be stored.
4. Name of the machine from which to download the cookie.
5. Date and time when the cookie expires.

GetFullCookieList()

Returns the number of cookies and returns (by reference) an array of cookies.

DeleteCookie()

Deletes a cookie from the request.

Remember that cookies are specific to an HTTP server. When you insert a cookie, you are using a connection to a specific server, and the cookie is not available on other servers.

For details on these methods, see the class documentation for %Net.HttpRequest.

1.10 Sending the HTTP Request

After you have created the HTTP request, use one of the following methods to send it:

Get()

```
method Get(location As %String = "",
           test As %Integer = 0,
           reset As %Boolean = 1) as %Status
```

Issues the HTTP GET request. This method causes the web server to return the page requested.

Head()

```
method Head(location As %String,
            test As %Integer = 0,
            reset As %Boolean = 1) as %Status
```

Issues the HTTP HEAD request. This method causes the web server to return just the headers of the response and none of the body.

Post()

```
method Post(location As %String = "",
            test As %Integer = 0,
            reset As %Boolean = 1) as %Status
```

Issues the HTTP POST request. Use this method to send data to the web server such as the results of a form, or upload a file. For an example, see “Sending Form Data.”

Put()

```
method Put(location As %String = "",
           test As %Integer = 0,
           reset As %Boolean = 1) as %Status
```

Issues the HTTP PUT request. Use this method to upload data to the web server. PUT requests are not common.

Send()

```
method Send(type As %String,
            location As %String,
            test As %Integer = 0,
            reset As %Boolean = 1) as %Status
```

Sends the specified type of HTTP request to the server. This method is normally called by the other methods, but is provided for use if you want to use a different HTTP verb. Here *type* is a string that specifies an HTTP verb such as `"POST"`.

In all cases:

- Each method returns a status, which you should check.
- If the method completes correctly, the response to this request will be in the *HttpResponse* property.
- The *location* argument is the URL to request, for example: `"/test.html"`.
- The *location* argument can contain parameters, which are assumed to be already URL-escaped, for example: `"/test.html?PARAM=%25VALUE"` sets PARAM equal to %VALUE.
- Use the *test* argument to check that you are sending what you are expecting to send:
  - If *test* is 1 then instead of connecting to a remote machine, the method will just output what it would have send to the web server to the current device.
  - If *test* is 2 then it will output the response to the current device after issuing the HTTP request.
- Each method automatically calls the **Reset()** method after reading the response from the server, except if *test*=1 or if *reset*=0.

  The **Reset()** method resets the %Net.HttpRequest instance so that it can issue another request. This is much faster than closing this object and creating a new instance. This also moves the value of the `Location` header to the `Referer` header.

For example:

ObjectScript

```
Set httprequest=##class(%Net.HttpRequest).%New()
Set httprequest.Server="www.intersystems.com"
Do httprequest.Get("/")
```

For other examples, see the class documentation for %Net.HttpRequest.

1.11 Creating and Sending Multipart POST Requests

To create and send a multipart POST request, use the %Net.MIMEPart classes, which are discussed more fully later in this book. The following example sends a POST request with two parts. The first part includes file binary data, and the second part includes the file name.

Class Member

```
ClassMethod CorrectWriteMIMEMessage3(header As %String)
{
     // Create root MIMEPart
     Set RootMIMEPart=##class(%Net.MIMEPart).%New()

     //Create binary subpart and insert file data
     Set BinaryMIMEPart=##class(%Net.MIMEPart).%New()
     Set contentdisp="form-data; name="_$CHAR(34)_"file"_$CHAR(34)_"; filename="
                     _$CHAR(34)_"task4059.txt"_$CHAR(34)
     Do BinaryMIMEPart.SetHeader("Content-Disposition",contentdisp)

     Set stream=##class(%FileBinaryStream).%New()
     Set stream.Filename="/home/tabaiba/prueba.txt"
     Do stream.LinkToFile("/home/tabaiba/prueba.txt")

     Set BinaryMIMEPart.Body=stream
```

```
     Do BinaryMIMEPart.SetHeader("Content-Type","text/plain")

    // Create text subpart
    Set TextMIMEPart=##class(%Net.MIMEPart).%New()
    Set TextMIMEPart.Body=##class(%GlobalCharacterStream).%New()
    Do TextMIMEPart.Body.Write("/home/tabaiba/prueba.txt")

    // specify some headers
    Set TextMIMEPart.ContentType="text/plain"
    Set TextMIMEPart.ContentCharset="us-ascii"
    Do TextMIMEPart.SetHeader("Custom-header",header)

    // Insert both subparts into the root part
    Do RootMIMEPart.Parts.Insert(BinaryMIMEPart)
    Do RootMIMEPart.Parts.Insert(TextMIMEPart)

    // create MIME writer; write root MIME message
    Set writer=##class(%Net.MIMEWriter).%New()

    // Prepare outputting to the HttpRequestStream
    Set SentHttpRequest=##class(%Net.HttpRequest).%New()
    Set status=writer.OutputToStream(SentHttpRequest.EntityBody)
    if $$$ISERR(status) {do $SYSTEM.Status.DisplayError(status) Quit}

    // Now write down the content
    Set status=writer.WriteMIMEBody(RootMIMEPart)
    if $$$ISERR(status) {do $SYSTEM.Status.DisplayError(status) Quit}

    Set SentHttpRequest.Server="congrio"
    Set SentHttpRequest.Port = 8080

    Set ContentType= "multipart/form-data; boundary="_RootMIMEPart.Boundary
    Set SentHttpRequest.ContentType=ContentType

    set url="alfresco/service/sample/upload.json?"
            _"alf_ticket=TICKET_caee62bf36f0ea5bd51194fce161f99092b75f62"

    set status=SentHttpRequest.Post(url,0)
    if $$$ISERR(status) {do $SYSTEM.Status.DisplayError(status) Quit}
}
```

1.12 Accessing the HTTP Response

After you send an HTTP request, the HttpResponse property of the request is updated. This property is an instance of %Net.HttpResponse. This section describes how to use the response object. It includes the following topics:

- Accessing the Data of the Response
- Getting an HTTP Header by Name
- Accessing Other Information about the Response

For details, see the class documentation for %Net.HttpRequest.

1.12.1 Accessing the Data of the Response

The body of the HTTP response is contained in the Data property of the response. This property contains a stream object (specifically %GlobalBinaryStream). To work with this stream, use the standard stream methods: **Write()**, **WriteLine()**, **Read()**, **ReadLine()**, **Rewind()**, **MoveToEnd()**, and **Clear()**. You can also use the Size property of the stream.

The ReadRawMode property of the *request* controls how the body of the response is read.

- By default, this property is false and Caché assumes that the body is in the character set specified in the HTTP headers of the response (and translates the character set accordingly).
- If this property is true, Cache reads the body in RAW mode (performing no translation of the character set).

You can also use the **OutputToDevice()** method, which writes the full response to the current device. The headers are not in the same order as generated by the web server.

The following shows a simple example in which we copy the response stream to a file and save it:

ObjectScript

```
set request=##class(%Net.HttpRequest).%New()
set request.Server="tools.ietf.org"
set request.Https=1
set request.SSLConfiguration="TEST"
set status=request.Get("/html/rfc7158")
if $$$ISERR(status) {
        do $system.OBJ.DisplayError()
} else {
        set response=request.HttpResponse
}

Set file=##class(%FileCharacterStream).%New()
set file.Filename="c:/temp/rfc7158.html"
set status=file.CopyFrom(response.Data)
if $$$ISERR(status) {
        do $system.OBJ.DisplayError()
}
do file.%Close()
```

1.12.2 Getting an HTTP Header by Name

The %Net.HttpResponse class stores its HTTP headers in a Caché multidimensional array. To access the headers, use the following methods:

GetHeader()

Returns the value of the given header.

GetNextHeader()

Returns the name of the next header after the given header.

Each of these methods takes a single argument, a string that is the name of an HTTP header.

You can also use the **OutputHeaders()** method, which writes the HTTP headers to the current device (although not in the same order they were generated).

1.12.3 Accessing Other Information about the Response

The %Net.HttpResponse class provides properties that store other specific parts of the HTTP response:

- StatusLine stores the HTTP status line, which is the first line of the response.
- StatusCode stores the HTTP status code.
- ReasonPhrase stores the human-readable reason that corresponds to StatusCode.
- ContentInfo stores additional information about the response body.
- ContentType stores the value of the `Content-Type:` header.
- HttpVersion indicates the version of HTTP that is supported by the web server that sent the response.

2

Sending and Receiving Email

This chapter describes how you can use Caché to send and receive MIME email messages. It discusses the following topics:

- Supported Email Protocols
- How Caché Represents MIME Email Messages
- Creating Single-part Email Messages
- Creating Multipart Email Messages
- Specifying Email Message Headers
- Adding Attachments to a Message
- Using an SMTP Server to Send Email
- Fetching Email from a POP3 Server
- Working with a Received Email Message
- Automatic Encoding and Character Translation

Also see the class documentation for examples and extensive comments.

Note: The examples in this chapter are organized so that the methods for managing email messages can be used with different email servers, which is useful during testing and demonstrations. This is not necessarily the code organization that is most suitable for production needs.

2.1 Supported Email Protocols

Email sends messages across the Internet using standard protocols. Caché supports three of these protocols, as follows:

- Caché provides an object representation of MIME email messages. It supports text and non-text attachments, single-part or multipart message bodies, and headers in ASCII and non-ASCII character sets. (For more general support of MIME parts, see the chapter "Creating and Writing MIME Messages.")
- You can send email via an SMTP server. SMTP (Simple Mail Transport Protocol) is the Internet standard for sending email.
- You can also retrieve email from an email server via POP3, the most common standard for retrieving email from remote servers.

Note: Caché does not provide a mail server. Instead, it provides the ability to connect to and interact with mail servers.

2.2 How Caché Represents MIME Email Messages

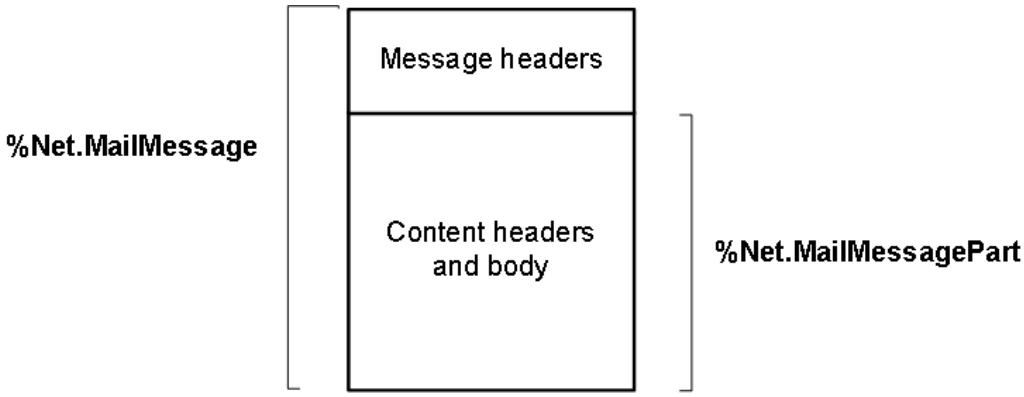
First, it is useful to understand how Caché represents MIME email messages.

In general, a multipart MIME message consists of the following pieces:

- A set of *message headers*, each of which contains information such as the address to which the message is sent. This also includes the `Mime-Type` header and the `Content-Type` header for the entire message.

  For a multipart message, the `Content-Type` header must be `multipart/mixed` or some other subtype of `multipart`; the MIME standard has many variants.
- Multiple *message parts*, each of which consists of the pieces:
  - A set of *content headers*, which includes the `Content-Type` header and other headers specific to this part.
  - A *body*, which is either text or binary, and which can be in a different character set than the bodies of other parts.

Caché uses two classes to represent email messages: %Net.MailMessage and %Net.MailMessagePart, the superclass of %Net.MailMessage. The following graphic shows the relationship between these classes:



In general:

- To represent an ordinary, one-part message, you use %Net.MailMessage
- To represent a multipart message, you use %Net.MailMessage as the parent message and you use multiple instances of %Net.MailMessagePart as its parts.

The following sections provide details.

2.3 Creating Single-part Email Messages

To create a single-part email message, you use the %Net.MailMessage class. To create a mail message, do the following:

1. Create an instance of %Net.MailMessage.

Tip: You can specify a character set as the argument to **%New()**; if you do so, that sets the Charset property for the message. For information on how this affects the message, see "Automatic Encoding and Character Translation."

2. Set the To, From, and Subject properties of your instance.
   - To — The list of email addresses to which this message will be sent. This property is a standard Caché list class; to work with it, you use the standard list methods: **Insert()**, **GetAt()**, **RemoveAt()**, **Count()**, and **Clear()**.
   - From — The email address this message is sent from.
   - Subject — The subject of the message, if this is required by the SMTP server you are using.
3. Optionally set Date, Cc, Bcc, and other properties. For details, see "Specifying Basic Email Headers."
4. If the message is not plain text, set the following properties to indicate the kind of message you are creating:
   - If this is an HTML message, set the IsHTML property to 1.
   - If this is a binary message, set the IsBinary property to 1.
5. To specify the character set of the message and its headers, set the Charset property as needed. (For details on how this affects the message, see "Automatic Encoding and Character Translation.")

   Important: It is important to specify the character set *before* you add the contents of the message.
6. Add the contents of the message:
   - For plain text or HTML, use the TextData property, which is an instance of %FileCharacterStream. You do not need to specify the TranslateTable property of this stream; that occurred automatically when you specified the character set of the mail message.
   - For binary data, use the BinaryData property, which is an instance of %FileBinaryStream.

   Tip: When you specify the Filename property of the stream, be sure to use a directory to which the users will have write access.

   To work with these properties, use the standard stream methods: **Write()**, **WriteLine()**, **Read()**, **ReadLine()**, **Rewind()**, **MoveToEnd()**, and **Clear()**. You can also use the Size property of the stream, which gives you the size of the message contents.

Note: You should be aware of the requirements of the SMTP server that you are using. For example, some SMTP servers require that you include a `Subject` header. Similarly, some SMTP servers do not permit arbitrary `From` headers.

Similarly, some SMTP servers recognize the `Priority` header and others recognize `X-Priority` instead.

Also see "Creating Multipart Email Messages."

2.3.1 Example 1: CreateTextMessage()

The following method creates a simple message and specifies the addresses for it:

Class Member

```
ClassMethod CreateTextMessage() As %Net.MailMessage
{
    Set msg = ##class(%Net.MailMessage).%New()
    set msg.From = "test@test.com"
    Do msg.To.Insert("xxx@xxx.com")
    Do msg.Cc.Insert("yyy@yyy.com")
    Do msg.Bcc.Insert("zzz@zzz.com")
    Set msg.Subject="subject line here"
    Set msg.IsBinary=0
    Set msg.IsHTML=0
    Do msg.TextData.Write("This is the message.")

    Quit msg
}
```

2.3.2 Example 2: SimpleMessage()

You may instead prefer to specify the addresses when you actually send the message (see "Example 3: SendMessage()" in "Using an SMTP Server to Send Email"). The following variation of the preceding example generates a text message with no addresses:

Class Member

```
ClassMethod SimpleMessage() As %Net.MailMessage
{
  Set msg = ##class(%Net.MailMessage).%New()
  Set msg.Subject="Simple message "_$h
  Set msg.IsBinary=0
  Set msg.IsHTML=0
  Do msg.TextData.Write("This is the message.")
  Quit msg
}
```

There are other examples in the SAMPLES namespace. To find them, search for `%Net.MailMessage` in that namespace.

2.4 Creating Multipart Email Messages

To create a multipart email message:

1. Create an instance of %Net.MailMessage and set its To, From, and Subject properties. Optionally set other properties to specify other message headers.
2. Set the IsMultiPart property to 1.
3. Set the MultiPartType property to one of the following: `"related"`, `"alternative"`, or `"mixed"`. This affects the `Content-Type` header of the entire message.
4. For each part that the message should contain, create an instance of %Net.MailMessagePart and specify its properties as described in "Creating Single-part Email Messages" — starting with step 4.
5. For the parent email message, set the Parts property, which is an array. Insert each child message part into this array.

When you send the message, the %Net.SMTP class automatically sets the `Content-Type` header for the message as appropriate (given the value of the MultiPartType property).

2.5 Specifying Email Message Headers

As noted previously, both the message itself and each part of a message has a set of headers.

The %Net.MailMessage and %Net.MailMessagePart classes provide properties that give you easy access to the most commonly used headers, but you can add any header you need. This section provides information on all the headers as well as how to create custom headers.

The headers of a given message part are in the character set specified by the Charset property of that part.

Note: You should be aware of the requirements of the SMTP server that you are using. For example, some SMTP servers require that you include a `Subject` header. Similarly, some SMTP servers do not permit arbitrary `From` headers.

Similarly, some SMTP servers recognize the `Priority` header and others recognize `X-Priority` instead.

2.5.1 Specifying Basic Email Headers

Set the following properties (only in %Net.MailMessage) to set the most commonly used headers of the message itself:

- To — (Required) The list of email addresses to which this message will be sent. This property is a standard Caché list; to work with it, you use the standard list methods: **Insert()**, **GetAt()**, **RemoveAt()**, **Count()**, and **Clear()**.
- From — (Required) The email address this message is sent from.
- Date — The date of this message.
- Subject — (Required) A string containing the subject for this message.
- Sender — The actual sender of the message.
- Cc — The list of carbon copy addresses to which this message will be sent.
- Bcc — The list of blind carbon copy addresses to which this message will be sent.

2.5.2 Content-Type Header

When you send the message, the `Content-Type` header for the message and for each message part is automatically set as follows:

- If the message is plain text (IsHTML equals 0 and IsBinary equals 0), the `Content-Type` header is set to `"text/plain"`.
- If the message is HTML (IsHTML equals 1 and IsBinary equals 0), the `Content-Type` header is set to `"text/html"`.
- If the message is binary (IsBinary equals 1), the `Content-Type` header is set to `"application/octet-stream"`.
- If the message is multipart, the `Content-Type` header is set as appropriate for the value of the MultiPartType property.

Both %Net.MailMessage and %Net.MailMessagePart provide the ContentType property, which gives you access to the `Content-Type` header.

2.5.3 Content-Transfer-Encoding Header

Both %Net.MailMessage and %Net.MailMessagePart provide the ContentTransferEncoding property, which provides an easy way to specify the `Content-Transfer-Encoding` header of the message or the message part.

This property can be one of the following: `"base64"` `"quoted-printable"` `"7bit"` `"8bit"`

The default is as follows:

- For a binary message or message part: "base64"

  Important: Note that if the content is "base64" encoded, it cannot contain any Unicode characters. If the content you wish to send includes Unicode characters, then make sure to use $ZCONVERT to convert the content to UTF-8, and then base-64 encode it. For example:

  ```
  set BinaryText=$ZCONVERT(UnicodeText,"O","UTF8")
  set Base64Encoded=$system.Encryption.Base64Encode(BinaryText)
  ```

  The recipient must use the reverse process to decode the text:

  ```
  set BinaryText=$system.Encryption.Base64Decode(Base64Encoded)
  set UnicodeText=$ZCONVERT(BinaryText,"I","UTF8")
  ```

- For a text message or message part: "quoted-printable"

Also see "Automatic Encoding and Character Translation."

2.5.4 Custom Headers

With both %Net.MailMessage and %Net.MailMessagePart, you can set or get custom headers by accessing the Headers property, which is an array with the following structure:

Array Key	Array Value
Name of the header, such as "Priority"	Value of the header

You use this property to contain additional headers such as X-Priority and others. For example:

ObjectScript

```
do msg.Headers.SetAt(1,"X-Priority")
do msg.Headers.SetAt("High","X-MSMail-Priority")
do msg.Headers.SetAt("High","Importance")
```

Different email servers and clients recognize different headers, so it can be useful to set multiple similar headers to be sure that the server or client receives a message with a header it can recognize.

2.6 Adding Attachments to a Message

You can add attachments to an email message or message part (specifically, to an instance of %Net.MailMessagePart or %Net.MailMessage). To do so, use the following methods:

Each of these methods adds the attachment to the Parts array of the original message (or message part), and automatically sets the IsMultiPart property to 1.

AttachFile()

```
method AttachFile(Dir As %String,
                  File As %String,
                  isBinary As %Boolean = 1,
                  charset As %String = "",
                  ByRef count As %Integer) as %Status
```

Attaches the given file to the email message. By default the file is sent as a binary attachment, but you can specify instead that it is text. You can also specify the character set that the file uses if it is text.

Specifically, this method creates an instance of %Net.MailMessagePart and places the contents of the file in the BinaryData or TextData property as appropriate, and sets the Charset property and TextData.TranslateTable properties if needed. The method returns, by reference, an integer that indicates the position of this new message part within the Parts array.

This method also sets the Dir and FileName properties of the message or message part.

AttachStream()

```
method AttachStream(stream As %Stream.Object,
                    Filename As %String,
                    isBinary As %Boolean = 1,
                    charset As %String = "",
                    ByRef count As %Integer) as %Status
```

Attaches the given stream to the email message. The attachment is considered a file attachment if *Filename* is specified. Otherwise it is considered an inline attachment. See the comments for **AttachFile()**.

AttachNewMessage()

```
method AttachNewMessage() as %Net.MailMessagePart
```

Creates a new instance of %Net.MailMessage, adds it to the message, and returns the newly modified parent message or message part.

AttachEmail()

```
method AttachEmail(mailmsg As %Net.MailMessage)
```

Given an email message (an instance of %Net.MailMessage), this method adds it to the message. This method also sets the Dir and FileName properties of the message or message part.

Note: This method sets ContentType to `"message/rfc822"`. In this case, you cannot add any other attachments.

2.6.1 Example: MessageWithAttachment()

The following example generates a simple email message with one hardcoded attachment. It does not provide any addresses for the message; you can provide that information when you actually send the message (see "Example 3: SendMessage()" in "Using an SMTP Server to Send Email").

Class Member

```
ClassMethod MessageWithAttachment() As %Net.MailMessage
{
  Set msg = ##class(%Net.MailMessage).%New()
  Set msg.Subject="Message with attachment "_$h
  Set msg.IsBinary=0
  Set msg.IsHTML=0
  Do msg.TextData.Write("This is the main message body.")

  //add an attachment
  Set status=msg.AttachFile("c:\", "GNET.pdf")
  If $$$ISERR(status) {
    Do $System.Status.DisplayError(status)
    Quit $$$NULLOREF
  }

  Quit msg
}
```

For other examples, see the class reference for the %Net.MailMessagePart class.

2.7 Using an SMTP Server to Send Email

If you have access to an SMTP server, you can send email messages. The SMTP server must be running and you must have the needed permissions to use it. To send email, do the following:

1. Create an instance of %Net.SMTP and set its properties as needed, especially the following:
   - smtpserver is the name of the SMTP server you are using.
   - port is the port you are using on the SMTP server; the default is 25.
   - timezone specifies the time zone of the server, as specified by RFC 822, for example `"EST"` or `"-0400"` or `"LOCAL"`. If this is not set, the message uses universal time.

   This object describes the SMTP server you will use.
2. If the SMTP server requires authentication, specify the necessary credentials. To do so:
   a. Create an instance of %Net.Authenticator.
   b. Set the UserName and Password properties of this object.
   c. Set the authenticator property of your %Net.SMTP instance equal to this object.
   d. If the message itself has an authorized sender, set the AuthFrom property of your %Net.SMTP instance.
3. To use an SSL/TLS connection to the SMTP server:
   a. Set the SSLConfiguration property to the name of the activated SSL/TLS configuration to use.

      For information on creating and managing SSL/TLS configurations, see "Using SSL/TLS with Caché" in the *Caché Security Administration Guide*. The SSL/TLS configuration includes an option called **Configuration Name**, which is the string to use in this setting.
   b. Set the UseSTARTTLS property to either 0 or 1.

      In most cases, use the value 0. Use the value 1 for the case in which the server interaction begins on a normal TCP socket and then switches to TLS on the same port as the normal socket. For details, see RFC 3207.
   c. Optionally set the SSLCheckServerIdentity property to 1. Do this if you want to verify the host server name in the certificate.
4. Create the email message to send (as described in "Creating Single-part Email Messages" and "Creating Multipart Email Messages").
5. Call the **Send()** method of your SMTP instance. This method returns a status, which you should check.
6. If the returned status indicates an error, check the Error property, which contains the error message itself.
7. Check the FailedSend property, which contains a list of email addresses for which the send action failed.

The examples in the following sections use a couple of different free SMTP services that were available at the time this manual was written. No particular endorsement is implied by the selection of these services. Also note that the examples do not show the actual passwords.

There are other examples in the SAMPLES namespace. To find them, search for `%Net.SMTP` in that namespace. Also see the class documentation for %Net.SMTP.

Important: %Net.SMTP writes the message body into a temporary file stream. By default, this file is written to the namespace directory and if the directory requires special write permissions, the file is not created and you get an empty message body.

You can define a new path for these temporary files and choose a path that does not restrict write access (for example, /tmp). To do so, set the global node `%SYS("StreamLocation",`*namespace*`)` where *namespace* is the namespace in which your code is running. For example:

```
Set ^%SYS("StreamLocation","SAMPLES")="/tmp"
```

If `%SYS("StreamLocation",`*namespace*`)` is null, then Caché uses the directory specified by `%SYS("TempDir",`*namespace*`)`. If `%SYS("TempDir",`*namespace*`)` is not set, then Caché uses the directory specified by `%SYS("TempDir")`

2.7.1 Example 1: HotPOPAsSMTP() and SendSimpleMessage()

This example consists of two methods that you use together. The first creates an instance of %Net.SMTP that uses a test account that has already been set up on the HotPOP SMTP server:

Class Member

```
ClassMethod HotPOPAsSMTP() As %Net.SMTP
{
  Set server=##class(%Net.SMTP).%New()
  Set server.smtpserver="smtp.hotpop.com"
  //HotPOP SMTP server uses the default port (25)
  Set server.port=25

  //Create object to carry authentication
  Set auth=##class(%Net.Authenticator).%New()
  Set auth.UserName="isctest@hotpop.com"
  Set auth.Password="123pass"

  Set server.authenticator=auth
  Set server.AuthFrom=auth.UserName
  Quit server
}
```

The next method sends a simple, unique message, using an SMTP server that you provide as the argument:

Class Member

```
ClassMethod SendSimpleMessage(server As %Net.SMTP) As %List
{
  Set msg = ##class(%Net.MailMessage).%New()
  Set From=server.authenticator.UserName
  Set:From="" From="xxx@xxx.com"
  Set msg.From = From

  Do msg.To.Insert("xxx@xxx.com")
  //Do msg.Cc.Insert("yyy@yyy.com")
  //Do msg.Bcc.Insert("zzz@zzz.com")
  Set msg.Subject="Unique subject line here "_$H
  Set msg.IsBinary=0
  Set msg.IsHTML=0
  Do msg.TextData.Write("This is the message.")

  Set status=server.Send(msg)
  If $$$ISERR(status) {
    Do $System.Status.DisplayError(status)
    Write server.Error
    Quit ""
  }
  Quit server.FailedSend
}
```

2.7.2 Example 2: YPOPsAsSMTP()

This example creates an instance of an instance of %Net.SMTP that uses YPOPs, which is client software that provides SMTP and POP3 access to a Yahoo email account. It uses a test account that has already been set up for this purpose:

Class Member

```
ClassMethod YPOPsAsSMTP() As %Net.SMTP
{
  Set server=##class(%Net.SMTP).%New()
  //local host acts as the server
  Set server.smtpserver="127.0.0.1"
  //YPOPs uses default port, apparently
  Set server.port=25

  //Create object to carry authentication
  Set auth=##class(%Net.Authenticator).%New()
  //YPOPs works with a Yahoo email account
  Set auth.UserName="isc.test@yahoo.com"
  Set auth.Password="123pass"

  Set server.authenticator=auth
  Set server.AuthFrom=auth.UserName
  Quit server
}
```

You can use this with the `SendSimpleMessage` method shown in the previous example.

2.7.3 Example 3: SendMessage()

The following, more flexible method accepts both an SMTP server and an email message. The email message should already include a subject line (if required by the SMTP server), but does not have to include addresses. This method then sends the email message to a set of hardcoded test destinations:

Class Member

```
ClassMethod SendMessage(server As %Net.SMTP, msg as %Net.MailMessage) as %Status
{
  Set From=server.authenticator.UserName
  //make sure From: user is same as used in authentication
  Set msg.From = From

  //finish addressing the message
  Do msg.To.Insert("xxx@xxx.com")
  //send the message to various test email addresses
  Do msg.To.Insert("isctest@hotpop.com")
  Do msg.To.Insert("isc_test@hotmail.com")
  Do msg.To.Insert("isctest001@gmail.com")
  Do msg.To.Insert("isc.test@yahoo.com")

  Set status=server.Send(msg)
  If $$$ISERR(status) {
    Do $System.Status.DisplayError(status)
    Write server.Error
    Quit $$$ERROR($$$GeneralError,"Failed to send message")
  }
  Quit $$$OK
}
```

This example is meant for use with the example methods `SimpleMessage` and `MessageWithAttachment` described in “Adding Attachments to a Message”.

2.7.4 Other Properties of %Net.SMTP

The %Net.SMTP class also has some other properties that you might need, depending on the SMTP server you are using:

- AllowHeaderEncoding specifies whether the **Send()** method encodes non-ASCII header text. The default is 1, which means that non-ASCII header text is encoded as specified by RFC 2047.

- ContinueAfterBadSend specifies whether to continue trying to send a message after detecting a failed email address. If ContinueAfterBadSend is 1, the system will add the failed email address to the list in the FailedSend property. The default is 0.
- ShowBcc specifies whether the Bcc headers are written to the email message. These will normally be filtered out by the SMTP server.

2.8 Fetching Email from a POP3 Server

This section discusses how to use the %Net.POP3 class to fetch email messages. It includes the following topics:

Also see the class documentation for %Net.FetchMailProtocol for examples and extensive comments. %Net.FetchMailProtocol is the abstract superclass of %Net.POP3.

2.8.1 Communicating with a POP3 Server

If you have the needed permissions and if the mail server is running, you can download and process email messages from it using the POP3 protocol. In general, to communicate with a POP3 server, you log in, perform a series of actions that affect a mailbox, and then either commit or roll back any changes. To do this in Caché:

1. Create an instance of %Net.POP3. This object describes the POP3 server you will use.
2. Optionally specify the following properties of your instance of %Net.POP3:
   - port — Specifies the port you will use; the default is 110.
   - timeout — Specifies the read timeout in seconds; the default is 30 seconds.
   - StoreAttachToFile — Specifies whether to save each attachment to a file, when a message is read (when the message includes the `content-disposition; attachment` header). The default is false.

     Note that this setting does nothing unless AttachDir is also set.
   - StoreInlineToFile — Specifies whether to save each inline attachment to a file, when a message is read (when the message includes the `content-disposition; inline` header). The default is false.

     Note that this setting does nothing unless AttachDir is also set.
   - AttachDir — Specifies the directory into which the attachment are saved. There is no default. Make sure to terminate the name of the directory with a slash (/) or backslash (\), as appropriate for the operating system. Also make sure that this is directory already exists, and the users have write access to it.
   - IgnoreInvalidBase64Chars — Specifies whether to ignore invalid characters found during base–64 decoding. The default is false (and invalid characters result in an error). Note that RFC 2045 is ambiguous about whether unexpected characters should be ignored or should result in an error during base–64 decoding.

3. To use an SSL/TLS connection to the POP3 server:
   a. Set the SSLConfiguration property to the name of the activated SSL/TLS configuration to use.

      For information on creating and managing SSL/TLS configurations, see "Using SSL/TLS with Caché" in the *Caché Security Administration Guide*. The SSL/TLS configuration includes an option called **Configuration Name**, which is the string to use in this setting.

   b. Set the UseSTARTTLS property to either 0 or 1.

      In most cases, use the value 0. Use the value 1 for the case in which the server interaction begins on a normal TCP socket and then switches to TLS on the same port as the normal socket. For details, see RFC 2595.

   c. Optionally set the SSLCheckServerIdentity property to 1. Do this if you want to verify the host server name in the certificate.

4. Call the **Connect()** method of your instance. This method takes three arguments, in order:
   a. The name of the POP3 server
   b. A username
   c. A password

5. Use the methods of your instance to examine the mailbox, retrieve messages, and delete messages. The following sections provide details.
6. Optionally, to prevent the connection from timing out, call the **Ping()** method of your %Net.POP3 instance.
7. Optionally, if you have marked messages for deletion but now choose not to delete them, call the **RollbackDeletes()** method of your %Net.POP3 instance.
8. When you are done making changes to the mailbox, call one of the following methods:
   - **QuitAndCommit()** — Commits your changes and logs out of the mail server.
   - **QuitAndRollback()** — Rolls back your changes and logs out of the mail server.

Each of these methods returns a status, which you should check before continuing. Also see the class reference for %Net.POP3 for complete method signatures.

The examples in the following sections use two different free POP3 services that were available at the time this manual was written. No particular endorsement is implied by the selection of these services. Also note that the examples do not show the actual passwords.

2.8.1.1 Example 1: HotPOPAsPOP3()

The following method logs into the HotPOP POP3 server using an account that was previously set up for this purpose:

Class Member

```
ClassMethod HotPOPAsPOP3() As %Net.POP3
{
  Set server=##class(%Net.POP3).%New()

  //HotPOP POP3 server uses the default port
  //but let's set it anyway
  Set server.port=110

  //just in case we plan to fetch any messages
  //that have attachments
  Set server.StoreAttachToFile=1
  Set server.StoreInlineToFile=1
  Set server.AttachDir="c:\DOWNLOADS\"

  Set servername="pop.hotpop.com"
```

```
  Set user="isctest@hotpop.com"
  Set pass="123pass"

  Set status=server.Connect(servername,user,pass)
  If $$$ISERR(status) {
    Do $System.Status.DisplayError(status)
    Quit $$$NULLOREF
  }
  Quit server
}
```

This method returns the %Net.POP3 server instance. Many of the examples later in this chapter accept the %Net.POP3 instance as an argument.

2.8.1.2 Example 2: YPOPsAsPOP3()

The following method also returns a %Net.POP3 server instance. In this case, we are using YPOPs, which is client software that provides SMTP and POP3 access to a Yahoo email account. It uses a test account that has already been set up for this purpose:

Class Member

```
ClassMethod YPOPsAsPOP3() As %Net.POP3
{
  Set server=##class(%Net.POP3).%New()

  //YPOPs uses the default port
  //but let's set it anyway
  Set server.port=110

  //just in case we plan to fetch any messages
  //that have attachments
  Set server.StoreAttachToFile=1
  Set server.StoreInlineToFile=1
  Set server.AttachDir="c:\DOWNLOADS\"

  //local host acts as the server
  Set servername="127.0.0.1"
  //YPOPs works with a Yahoo email account
  Set user="isc.test@yahoo.com"
  Set pass="123pass"

  Set status=server.Connect(servername,user,pass)
  If $$$ISERR(status) {
    Do $System.Status.DisplayError(status)
    Quit $$$NULLOREF
  }
  Quit server
}
```

2.8.2 Getting Information about the Mailbox

While you are connected to a POP3 server, you are logged into a user account and have access to the mailbox for that user account. Use the following methods to find what the mailbox contains:

GetMailBoxStatus()

Returns, by reference, the number of messages in the mailbox and the number of bytes that the mailbox uses.

GetMessageUIDArray()

If given an empty string as the first argument, this method returns, by reference, an array of information about the messages in the mailbox (excluding any that are currently marked for deletion). Each element in this array contains the following information about one message:

Array Key	Array Item
Number of the message, within the mailbox in its current state. The first message is number 1, and so on. The message number of a given message is not guaranteed to be the same in all sessions.	Unique message identifier (UID), which is the permanent identifier of this message available in all sessions. UIDs are unique to each mailbox.

GetSizeOfMessages()

If given an empty string as the first argument, this method returns, by reference, an array of information about the messages in the mailbox (excluding any that are currently marked for deletion). Each element in this array contains the following information about one message:

Array Key	Array Item
Number of the message, within the mailbox in its current state.	Size of this message, in bytes.

Each of these methods returns a status, which you should check before continuing. Also see "Other Message Retrieval Methods" for more details on these methods.

2.8.2.1 Example: ShowMailbox()

For example, the following method writes information about the mailbox that we are currently accessing:

Class Member

```
ClassMethod ShowMailbox(server as %Net.POP3)
{
    Set status=server.GetMailBoxStatus(.count,.size)
    If $$$ISERR(status) {
       Do $System.Status.DisplayError(status)
       Quit
    }
    Write "Mailbox information *****",!
    Write "Number of messages in mailbox: ",count,!
    Write "Size of messages: ",size,!

    Set status=server.GetMessageUIDArray(,.uids)
    Set status=server.GetSizeOfMessages(,.sizes)

    //iterate through messages, get info, and write it
    For i=1:1:count {
        Set uid=uids.GetAt(i)
        Set size=sizes.GetAt(i)
        Write "Msg number:", i,"   UID:",uid, "   size:",size,!
    }

}
```

This method generates output similar to the following:

```
Mailbox information *****
Number of messages in mailbox: 4
Size of messages: 18634
Msg number:1   UID:6ef78df6fd660391   size:7245
Msg number:2   UID:7410041a6faf4a87   size:5409
Msg number:3   UID:5555af7fa489e406   size:5121
Msg number:4   UID:299ad2b54c01a6be   size:859
```

2.8.3 Fetching Messages from the Mailbox

To simply get a message, use one of the following methods of the %Net.POP3 class:

Fetch()

Given a message number as the first argument, this method returns (by reference, as the second argument) an instance of %Net.MailMessage that contains that message.

FetchMessage()

Given a message number as the first argument, this method returns (by reference) information such as the `From` and `To` and other common headers, an array containing *all* the headers (including the common ones), and the message contents themselves

Each of these methods returns a status, which you should check before continuing. Note that these methods return an error status if the message is currently marked for deletion.

Also see "Other Message Retrieval Methods," which shows the complete method signatures for **Fetch()** and **FetchMessage()**

2.8.3.1 Example: FetchMailbox()

The following example is a variation of the `ShowMailbox` example described in "Getting Information about the Mailbox". This method uses the **Fetch()** method, examines each message, and writes the subject line of each message:

Class Member

```
ClassMethod FetchMailbox(server As %Net.POP3)
{
  Set status=server.GetMailBoxStatus(.count,.size)
  If $$$ISERR(status) {
    Do $System.Status.DisplayError(status)
    Quit $$$NULLOREF
  }
  Write "Mailbox information *****",!
  Write "Number of messages in mailbox: ",count,!
  Write "Size of messages: ",size,!

  Set status=server.GetMessageUIDArray(,.uids)
  Set status=server.GetSizeOfMessages(,.sizes)

  //iterate through messages, get info, and write it
  For i=1:1:count {
    Set uid=uids.GetAt(i)
    Set size=sizes.GetAt(i)
    Set status=server.Fetch(i,.msg)
    If $$$ISERR(status) {
      Set subj="***error***"
    } else{
      Set subj=msg.Subject
    }
    Write "Msg number:", i,"  UID:",uid, "  Size:",size
    Write "  Subject: ",subj,!
  }
}
```

2.8.4 Saving Attachments as Files

The `Content-Disposition` header might specify `attachment`, with or without a filename. For example:

```
Content-Disposition: attachment; filename=genome.jpeg;
```

If the `Content-Disposition` header does specify `attachment`, your %Net.POP3 instance can save all attachments in the message to files. To make this happen:

1. Specify the following properties of your %Net.POP3 instance:
   - Specify StoreAttachToFile as 1.
   - Specify StoreInlineToFile as 1.

- Specify a valid directory for AttachDir. Make sure to terminate the name of the directory with a slash (/) or backslash (\), as appropriate for the operating system. Also make sure that this is directory already exists, and the users have write access to it.

2. Call **Fetch()** or **FetchMessage()** of your %Net.POP3 instance.

Each filename is determined as follows:

1. If the `Content-Disposition` header specifies a filename, that filename is used.
2. Otherwise, if the `Content-Type` header specifies a filename, that filename is used.
3. Otherwise, the system creates a name of the form ATTxxxxxx.dat.

Note the following points:

- If the file already exists, the attachment is not downloaded.
- There is no default for AttachDir.
- The size of the attachment is not limited by Caché but might be limited by the file system.
- The Dir and FileName properties are not used here. They are relevant only when you upload an attachment to a mail message, as described in "Adding Attachments to a Message".

2.8.4.1 Example: GetMsg()

The following example method retrieves an entire message, given an instance of %Net.POP3 and a message number:

Class Member

```
ClassMethod GetMsg(server as %Net.POP3,msgno as %Integer) as %Net.MailMessage
{
    Set status=server.Fetch(msgno,.msg)
    If $$$ISERR(status) {
       Do $System.Status.DisplayError(status)
       Quit $$$NULLOREF
    }
    Quit msg
}
```

If the message had attachments, and if you specified the StoreAttachToFile, StoreInlineToFile, and AttachDir properties of the %Net.POP3 server, those attachments would be saved to the given directory when you called this method.

2.8.5 Getting Attached Email Messages

While you are connected to a mailbox, you can download any email messages that are attached to the email messages in the inbox. To do so, use the **GetAttachedEmail()** method of your %Net.POP3 instance to retrieve the contents of the enclosed email.

Given an instance of %Net.MailMessagePart, this method returns a single-part message that has contents of that message part. Specifically, it returns (as an output parameter) an instance of %Net.MailMessage initialized with the data taken from the attached email message.

2.8.6 Other Message Retrieval Methods

This section lists all the methods of %Net.POP3 that you can use to examine and retrieve messages.

Fetch()

```
method Fetch(MessageNumber As %Integer,
             ByRef MailMsg As %Net.MailMessage,
             Delete As %Boolean = 0,
             messageStream As %BinaryStream) as %Status
```

Returns (by reference) the message indicated by *MessageNumber* and optionally marks the message for deletion. Note that this method returns an error status if the message is already marked for deletion.

If *messageStream* is specified, then the original message is written to this binary stream.

FetchFromStream()

```
method FetchFromStream(messageStream As %BinaryStream, ByRef Msg As %Net.MailMessage) as %Status
```

This method is for use when you specify the *messageStream* argument for **Fetch()**.

Retrieves a single email message from the given binary stream. *messageStream* must be a binary stream containing the message. The message is returned by reference in *Msg*. This could be a multipart message.

FetchMessage()

```
method FetchMessage(MessageNumber As %Integer,
                    ByRef From As %String,
                    ByRef To As %String,
                    ByRef Date As %String,
                    ByRef Subject As %String,
                    ByRef MessageSize As %Integer,
                    ByRef MsgHeaders As %ArrayOfDataTypes,
                    ByRef MailMsg As %Net.MailMessage,
                    Delete As %Boolean = 0) as %Status
```

Returns (by reference) specific message headers, the message size, the message header array, and the message itself and optionally marks the message for deletion. Note that this method returns an error status if the message is already marked for deletion.

FetchMessageHeaders()

```
method FetchMessageHeaders(MessageNumber As %Integer,
                           ByRef MsgHeadersArray As %String) as %Status
```

Given a message number, this method returns (by reference) an array containing all the headers of that message. This method returns an error status if the message is currently marked for deletion.

FetchMessageInfo()

```
method FetchMessageInfo(MessageNumber As %Integer,
                        Lines As %Integer,
                        ByRef From As %String,
                        ByRef To As %String,
                        ByRef Date As %String,
                        ByRef Subject As %String,
                        ByRef MessageSize As %Integer,
                        ByRef MsgHeaders As %ArrayOfDataTypes,
                        ByRef MessageText As %String) as %Status
```

Given a message number, this method returns (by reference) specific message headers, the message size, the message header array, and the given number of lines of text from this message. This method returns an error status if the message is currently marked for deletion.

GetAttachedEmail()

```
method GetAttachedEmail(msgpart As %Net.MailMessagePart,
       Output mailmsg As %Net.MailMessage) as %Status
```

Given a message part, this method returns (as an output parameter) a single-part email message that is initialized with the data from the message part.

GetMessageUID()

```
method GetMessageUID(MessageNumber As %Integer,
                     ByRef UniqueID As %String) as %Status
```

Returns, by reference, the UID of a message, given a message number. See the previous section for details on message numbers and UIDs. This method returns an error status if the message is currently marked for deletion.

GetMessageUIDArray()

```
method GetMessageUIDArray(MessageNumber As %String = "",
                          ByRef ListOfUniqueIDs As %ArrayOfDataTypes) as %Status
```

If given an empty string as the first argument, this method returns, by reference, an array of information about the messages in the mailbox (excluding any that are currently marked for deletion). Each element in this array contains the following information about one message:

Array Key	Array Item
Number of the message, within the mailbox in its current state. The first message is number 1, and so on. The message number of a given message is not guaranteed to be the same in all sessions.	Unique message identifier (UID), which is the permanent identifier of this message available in all sessions. UIDs are unique to each mailbox.

Or, given a message number, this method returns a one-element array that contains the UID of that message. In this case, the method returns an error status if the message is currently marked for deletion.

GetSizeOfMessages()

```
method GetSizeOfMessages(MessageNumber As %String = "",
                         ByRef ListOfSizes As %ArrayOfDataTypes) as %Status
```

If given an empty string as the first argument, this method returns, by reference, an array of information about the messages in the mailbox (excluding any that are currently marked for deletion). Each element in this array contains the following information about one message:

Array Key	Array Item
Number of the message, within the mailbox in its current state.	Size of this message, in bytes.

Or, given a message number, this method returns a one-element array that contains the size (in bytes) of that message. In this case, this method returns an error status if the message is currently marked for deletion.

2.8.7 Deleting Messages

While you are connected to a mailbox, you can mark messages for deletion in the mailbox that you are logged into. You can do this in a couple of ways.

- You can use the **DeleteMessage()** method. This method takes one argument, the message number to delete.
- When you retrieve a message with the **Fetch()** or **FetchMessage()** method, you can specify an optional argument that tells the POP3 server to mark the message for deletion after you have retrieved it.

Remember the following points:

- These methods do not delete a message; they mark it for deletion. The message is not deleted until you complete the POP3 transaction with **QuitAndCommit()**. If you simply disconnect from the server, your changes are discarded.
- You can call the **RollbackDeletes()** method to change the messages so that they are no longer marked for deletion.
- Each of these methods returns a status, which you should check.

2.8.7.1 Example: GetMsgAndDelete() and CommitChanges()

The following example retrieves a message and marks it for deletion:

Class Member

```
ClassMethod GetMsgAndDelete(ByRef server As %Net.POP3, msgno As %Integer) As %Net.MailMessage
{
  //third argument to Fetch says whether to
  //mark for deletion
  Set status=server.Fetch(msgno,.msg,1)
  If $$$ISERR(status) {
    Do $System.Status.DisplayError(status)
    Quit $$$NULLOREF
  }

  Quit msg
}
```

Note that this message returns (by reference) an altered version of the %Net.POP3; the altered version contains the information about which message is marked for deletion.

You would use the preceding method with a method like the following:

Class Member

```
ClassMethod CommitChanges(server As %Net.POP3) As %Status
{
  //commit all changes and log out
  Set status=server.QuitAndCommit()
  If $$$ISERR(status) {
    Do $System.Status.DisplayError(status)
    Quit $$$ERROR($$$GeneralError,"Failed to commit changes")
  }
  Quit $$$OK
}
```

Alternatively you would roll back the changes with **RollbackDeletes()** or **QuitAndRollback()**.

2.9 Working with a Received Email Message

This section describes how you can work with an email message (%Net.MailMessage) that you have retrieved via %Net.POP3.

2.9.1 Message Basics

After you retrieve an email message (%Net.MailMessage), you generally start by determining what kind of message it is and how to read it; that is, whether it is a multipart message and whether the parts are binary. In this step, you can use the ContentType property. Or you can use the IsBinary, IsHTML, and IsMultiPart properties, which indirectly provide the same information as ContentType.

If the message is a multipart message, each part is an instance of %Net.MailMessagePart.

2.9.2 Message Headers

Both the message itself and each part of a message has a set of headers.

The %Net.MailMessage and %Net.MailMessagePart classes provide properties that give you easy access to the most commonly used headers. For example, %Net.MailMessage provides properties such as To, From, Subject, and Date. The Headers array property lets you access any custom header; see "Specifying Email Message Headers."

Also, if you have retrieved a message via %Net.POP3, you can use the **GetAttribute()** method. Given a header name and an attribute, this method returns the value of that attribute.

2.9.3 Message Contents

Once you know what the general message structure is, use the following techniques to retrieve the contents:

- For a multipart message, use the Parts property, which is an array of the parts. `Parts.Count()` gives you the number of parts. The key for each part is an integer, starting with 1. Use the **GetAt()** method to retrieve a given part. A message part is an instance of %Net.MailMessagePart.

  For information on the relationship of %Net.MailMessage and %Net.MailMessagePart, see "How Caché Represents MIME Email Messages."
- For a binary message (or message part), use the BinaryData property.
- For a text message (or message part), use the TextData property.
  - If IsHTML is 0, the TextData property is an ordinary text string.
  - If IsHTML is 1, the TextData property is an HTML text string.

Note that the email client that sends a message determines any wrapping in the message. The mail server has no control over this; nor does Caché.

2.9.4 Other Message Information

The MessageSize property indicates the total length of the message, apart from any attached email messages.

The following methods provide additional information about the message:

GetLocalDateTime()

Returns the date and time when the message was retrieved, converted to local time in **$HOROLOG** format.

GetUTCDateTime()

Returns the date and time when the message was retrieved, converted to UTC in **$HOROLOG** format.

GetUTCSeconds()

Returns the date and time when the message was retrieved, in seconds since 12/31/1840.

The following class methods are also available for time/date conversion:

HToSeconds()

A class method that converts a date/time in **$HOROLOG** format to seconds since 12/31/1840.

SecondsToH()

A class method that converts seconds since 12/31/1840 to a date/time in **$HOROLOG** format.

2.9.5 Example 1: ShowMsgInfo()

Given an instance of %Net.MailMessage, the following method writes information about the message to the current device:

Class Member

```
ClassMethod ShowMsgInfo(msg as %Net.MailMessage)
{
    Write "Message details *****",!
    Write "To (count): ", msg.To.Count(),!
    Write "From: ", msg.From,!
    Write "Cc (count): ", msg.Cc.Count(),!
    Write "Bcc (count): ", msg.Bcc.Count(),!
    Write "Date: ", msg.Date,!
    Write "Subject: ", msg.Subject,!
    Write "Sender: ", msg.Sender,!
    Write "IsMultipart: ", msg.IsMultiPart,!
    Write "Number of parts: ", msg.Parts.Count(),!
    Write "Number of headers: ", msg.Headers.Count(),!
    Write "IsBinary: ", msg.IsBinary,!
    Write "IsHTML: ", msg.IsHTML,!
    Write "TextData: ", msg.TextData.Read(),!
    Write "BinaryData: ", msg.BinaryData.Read(),!
}
```

This method produces output similar to the following:

```
Message details *****
To (count): 1
From: "XXX XXX" <XXX@XXX.com>
Cc (count): 0
Bcc (count): 0
Date: Fri, 16 Nov 2007 11:57:46 -0500
Subject: test 5
Sender:
IsMultipart: 0
Number of parts: 0
Number of headers: 16
IsBinary: 0
IsHTML: 0
TextData: This is test number 5, which is plain text.
BinaryData:
```

2.9.6 Example 2: ShowMsgPartInfo()

The following method writes information about a *part* of a message:

Class Member

```
ClassMethod ShowMsgPartInfo(msg as %Net.MailMessage, partno as %Integer)
{
    Set part=msg.Parts.GetAt(partno)
    Write "Message part details *****",!
    Write "Message part: ", partno,!
    Write "IsMultipart: ", part.IsMultiPart,!
    Write "Number of parts: ", part.Parts.Count(),!
    Write "Number of headers: ", part.Headers.Count(),!
    Write "IsBinary: ", part.IsBinary,!
    Write "IsHTML: ", part.IsHTML,!
    Write "TextData: ", part.TextData.Read(),!
    Write "BinaryData: ", part.BinaryData.Read(),!
}
```

This produces output similar to the following (given a different message than previously shown):

```
Message part details *****
Message part: 1
IsMultipart: 0
Number of parts: 0
Number of headers: 2
IsBinary: 0
IsHTML: 0
TextData: 1 test string


BinaryData:
```

2.9.7 Example 3: ShowMsgHeaders()

The following method writes information about the headers of a message; you could write a similar method that did the same for a message part.

Class Member

```
ClassMethod ShowMsgHeaders(msg as %Net.MailMessage)
{
    Set headers=msg.Headers
    Write "Number of headers: ", headers.Count(),!

    //iterate through the headers
    Set key=""
    For {
        Set value=headers.GetNext(.key)
        Quit:key=""
        Write "Header:",key,!
        Write "Value: ",value,!!
    }
}
```

This produces output similar to the following:

```
Number of headers: 16
Header: content-class
Value: urn:content-classes:message

Header: content-type
Value: multipart/alternative; boundary="----_=_NextPart_001_01C8286D.D9A7F3B1"

Header: date
Value: Fri, 16 Nov 2007 11:29:24 -0500

Header: from
Value: "XXX XXX" <XXX@XXX.com>

Header: message-id
Value: <895A9EF10DBA1F46A2DDB3AAF061ECD501801E86@Exchange1_backup>

Header: mime-version
Value: 1.0

...
```

2.10 Automatic Encoding and Character Translation

A email message part contains information about both the character sets used and the content-transfer-encoding used (if any). For reference, this section describes how this information is used.

For background information on character translation in Caché, see "Localization Support" in the *Caché Programming Orientation Guide*.

2.10.1 Outgoing Email

%Net.SMTP checks the Charset property of each part and then applies the appropriate translation table.

If you do not specify the Charset property for a given part, when you send the message, the following defaults are used for this part:

- On Unicode systems, the default is UTF-8.
- On 8–bit systems, the default is the system default character set.

%Net.SMTP also checks the ContentTransferEncoding property. If this property is `"base64"` or `"quoted-printable"`, then when it creates the message, %Net.SMTP encodes the body as needed. (If the content transfer encoding is `"7bit"` or `"7bit"`, no encoding is needed.)

Important: Note that if the content is `"base64"` encoded, it cannot contain any Unicode characters. If the content you wish to send includes Unicode characters, then make sure to use $ZCONVERT to convert the content to UTF-8.

```
set BinaryText=$ZCONVERT(UnicodeText,"O","UTF8")
set Base64Encoded=$system.Encryption.Base64Encode(BinaryText)
```

2.10.2 Incoming Email

%Net.POP3 checks the `Content-Transfer-Encoding` header of each message part and decodes the body as needed.

Then %Net.POP3 checks the `Content-Type` header of each message part. This affects the Charset property of the message part and also controls the translation table used when the message part is created in Caché.

3

Creating, Writing, and Reading MIME Messages

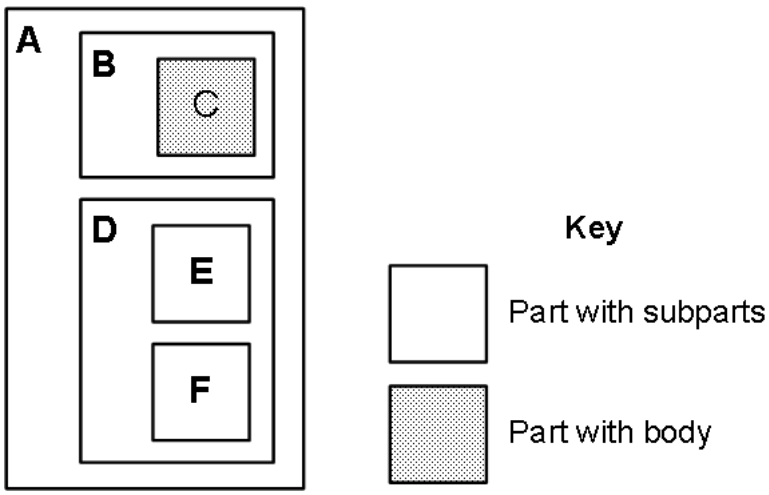
Caché provides a class that you can use to create multipart MIME messages (%Net.MIMEPart). You use this class when you create attachments to add to SOAP messages; see *Creating Web Services and Web Clients in Caché*. Because MIME is a common standard, there are many other possible applications, such as email processing and HTTP multipart POST.

This chapter discusses the following topics:

- An Overview of MIME Messages
- Creating MIME Parts
- Writing MIME Messages
- Reading MIME Messages

3.1 An Overview of MIME Messages

A document in MIME format is referred to as a *MIME part*. Each MIME part has headers and either contains a message body (either text or binary) or contains additional MIME parts. A MIME part that has a `MIME-Version` header can be used as a top-level document and is called a *MIME message*. The following figure shows an example:



In this example, E and F have additional subparts that are not shown.

To represent a MIME part, you use the %Net.MIMEPart class, which provides properties that you use to set the headers and contents of the part.

3.2 Creating MIME Parts

To create a MIME part, do the following:

1. Create an instance of %Net.MIMEPart.
2. Do one of the following:
   - Add a text or binary body. To do so, create an instance of a stream (either text or binary) and set the Body property of your MIME part equal to this stream. Use the standard stream interface to write data into this stream. Do not specify a value for the Parts property.
   - Add a list of MIME parts. To do so, create the MIME parts as described here and set the Parts property equal to a list of these parts. Do not specify a value for the Body property.
3. Optionally set headers as described in "Setting and Getting MIME Part Headers."

3.2.1 Setting and Getting MIME Part Headers

You can set values for and get values of the HTTP headers. The following properties of %Net.MIMEPart affect the MIME headers:

- ContentType — The Internet media type (MIME type) of the `Content-Type` header. This specifies the Internet media type of the Body data. For example: `"text/plain"`, `"text/html"`, `"image/jpeg"`, `"multipart/mixed"` and so on.
- ContentCharset — The `charset` part of the `Content-Type` header. If you set this, you must *first* set the ContentType property. For each MIME part that contains a text body, be sure to set the ContentCharset property appropriately to indicate the character set used in the body. This property should declare the character set that is already used, since %Net.MIMEPart does not perform any conversion.
- ContentId — The normalized `Content-ID` header, without the angle brackets (<>) and any leading and trailing spaces.
- ContentLocation — The normalized `Content-Location` header, without any leading and trailing spaces.
- ContentTransferEncoding — The `Content-Transfer-Encoding` header. This property can be one of the following: `"base64"` `"quoted-printable"` `"7bit"` `"8bit"`

  There is no default value.

  Important: Note that if the content is `"base64"` encoded, it cannot contain any Unicode characters. If the content you wish to send includes Unicode characters, then make sure to use $ZCONVERT to convert the content to UTF-8, and then base-64 encode it. For example:

  ```
  set BinaryText=$ZCONVERT(UnicodeText,"O","UTF8")
  set Base64Encoded=$system.Encryption.Base64Encode(BinaryText)
  ```

  The recipient must use the reverse process to decode the text:

  ```
  set BinaryText=$system.Encryption.Base64Decode(Base64Encoded)
  set UnicodeText=$ZCONVERT(BinaryText,"I","UTF8")
  ```

The %Net.MIMEPart class provides general methods that you can use to manage the MIME headers:

- **GetHeader()** returns the value of a header.
- **NextHeader()** gets the next header.
- **SetHeader()** sets the value of a header. Typically you use this to set nonstandard headers.
- **RemoveHeader()** removes a header.

For complete method signatures and other details, see the class documentation for %Net.MIMEPart.

3.2.2 Specifying an Optional Message Boundary Value

By default, message boundaries are generated automatically. You can specify the message boundary, if needed. To do so, specify a value for the Boundary property. Be sure to use a string that is extremely unlikely to be used in any of the message parts.

3.3 Writing MIME Messages

To write MIME messages, use %Net.MIMEWriter as follows:

1. Create an instance of the %Net.MIMEWriter class.
2. Optionally specify an output destination. To do so, use one of the following methods of your writer instance: **OutputToDevice()** (the default), **OutputToFile()**, or **OutputToStream()**.
3. Call methods of your writer to write output as needed:
   - Given a header name and value, **WriteHeader()** writes that header.
   - Given an instance of %Net.MIMEPart, **WriteMIMEBody()** writes the message body, which can have multiple parts.

     If the message is multipart, this method does not write any headers; it is your responsibility to write them. If the message is not multipart, however, the method does write the headers.
   - Given an instance of %Net.MIMEPart, **WriteMIMEMessage()** writes the MIME message, including all headers.

   For single-part messages, **WriteMIMEBody()** and **WriteMIMEMessage()** produce the same output.

For complete method signatures and other details, see the class documentation for %Net.MIMEPart.

3.3.1 Example: WriteMIMEMessage()

The following example demonstrates the use of **WriteMIMEMessage()**:

Class Member

```
ClassMethod WriteMIMEMessage(text As %String,header as %String) as %Status
{
 Set msg=##class(%Net.MIMEPart).%New()
 Set msg.Body=##class(%GlobalCharacterStream).%New()
 Do msg.Body.Write(text)

 //specify some headers
 Set msg.ContentType="text/html"
 Set msg.ContentCharset="us-ascii"
 Do msg.SetHeader("Custom-header",header)

 //create MIME writer; write MIME message
 Set writer=##class(%Net.MIMEWriter).%New()
 Set status=writer.WriteMIMEMessage(msg)

 If $$$ISERR(status) do $system.Status.DisplayError(status)
 Quit $$$OK
}
```

The following Terminal session shows this method in use:

```
GNET> Set text = "message text"

GNET> Set header="my header value"

GNET> Do ##class(GNET.MIME).WriteMIMEMessage(text,header)
CONTENT-TYPE: text/html
Custom-header: my header value

message text

GNET>
```

3.4 Reading MIME Messages

To read MIME messages, use %Net.MIMEReader, as follows:

1. Create an instance of the %Net.MIMEReader class.
2. Specify the source of input. To do so, use one of the following methods of your reader instance: **OpenFile()** or **OpenStream()**.
3. Call the **ReadMIMEMessage()** method of your reader instance. This method returns an instance of %Net.MIMEPart by reference as the first argument. It returns a status, which you should check.

For complete method signatures and other details, see the class documentation for %Net.MIMEPart.

4

Using FTP

Caché provides a class, %Net.FtpSession, that you can use to establish a session with an FTP server from within Caché. This chapter describes the following:

- Establishing an FTP Session
- FTP File and System Methods
- Using a Linked Stream to Upload Large Files
- Customizing Callbacks Issued by the FTP Server

4.1 Establishing an FTP Session

To establish an FTP session, do the following:

1. Create an instance of %Net.FtpSession.
2. Optionally set properties of this instance in order to control the general behavior of the session:
   - Timeout specifies how long to wait for a reply from the FTP server, in seconds.
   - SSLConfiguration specifies the activated SSL/TLS configuration to use for the connection, if any. Use this if the FTP server uses https.

     For information on creating and managing SSL/TLS configurations, see "Using SSL/TLS with Caché" in the *Caché Security Administration Guide*. The SSL/TLS configuration includes an option called **Configuration Name**; this is the string to use in this setting.
   - TranslateTable specifies the translation table to use when reading the contents of a file or writing the contents of a file.

     To find the name of the table for a given character set, use the %Net.Charset class described in "Other %Net Tools."
   - UsePASV enables PASV mode.
   - SSLCheckServerIdentity applies when the FTP server uses https. By default, when an instance of %Net.FtpSession connects to a SSL/TLS server, it checks whether the certificate server name matches the DNS name used to connect to the server. If these names do not match, the connection is not permitted. This default behavior prevents "man in the middle" attacks and is described in RFC 2818, section 3.1; also see RFC 2595, section 2.4.

     To disable this check, set the SSLCheckServerIdentity property to 0.

3. Call the **Connect()** method to connect to a specific FTP server.
4. Call the **Ascii()** or **Binary()** method to set the transfer mode to ASCII mode or binary mode, respectively. To see the current transfer mode, check the value of the Type property of your instance.

Note: Each method of %Net.FtpSession returns a status, which you should check. The methods also set the values of properties that provide useful information on the state of the session:

- Connected is true if you are currently connected, and is false otherwise.
- ReturnCode contains the return code from the last communication with the FTP server.
- ReturnMessage contains the return message from the last communication with the FTP server.

The **Status()** method returns (by reference) the status of the FTP server.

For details, see the class documentation for %Net.FtpSession.

4.1.1 Translate Table for Commands

%Net.FtpSession uses the technique described in RFC 2640 to automatically handle character set translation when looking at filenames and pathnames on an FTP server. When an instance of %Net.FtpSession connects to an FTP server, it uses the FEAT message to determine whether the server UTF-8 characters. If so, it switches the command channel communication to UTF-8 so that all filenames and pathnames will be correctly translated to and from UTF-8.

If the server does not support the FEAT command or does not report that it supports UTF-8, the %Net.FtpSession instance uses RAW mode and reads or writes the raw bytes.

In rare cases, if you need to specify the translation table to use, set the CommandTranslateTable property of the %Net.FtpSession instance. It should not be generally necessary to use this property.

4.2 FTP File and System Methods

Once you establish an FTP session, call methods of your session instance to perform FTP tasks. %Net.FtpSession provides the following methods for reading and writing files:

Delete()

Deletes a file.

Retrieve()

Copies a file from the FTP server into a Caché stream and returns the stream by reference. To work with this stream, use the standard stream methods: **Write()**, **WriteLine()**, **Read()**, **ReadLine()**, **Rewind()**, **MoveToEnd()**, and **Clear()**. You can also use the Size property of the stream.

RetryRetrieve()

Allows you to continue retrieving a file, given a stream created by a previous use of **Retrieve()**.

Store()

Writes the contents of a Caché stream into a file on the FTP server.

StoreFiles()

Given a local directory, and a wildcard mask, this method writes multiple files to that directory. Note that this method ignores directories, and uses the current transfer mode (binary or ASCII), which means that you cannot upload a mixed set of binary and ASCII files in one call.

Append()

Appends the contents of a stream to the end of the specified file.

Rename()

Renames a file.

In addition, %Net.FtpSession provides methods for navigating and modifying the file system on the FTP server: **GetDirectory()**, **SetDirectory()**,**SetToParentDirectory()**, and **MakeDirectory()**.

To examine the contents of the file system, use the **List()** or **NameList()** methods.

- **List()** creates a stream that contains a list of all the files whose names match a given pattern and returns this stream by reference.
- **NameList()** creates an array of filenames and returns this array by reference.

You can also use the **ChangeUser()** method to change to a different user; this is faster than logging out and logging in again. Use the **Logout()** method to log out.

The **System()** method returns (by reference) information about the type of computer that is hosting the FTP server.

For details, see the class documentation for %Net.FtpSession.

4.3 Using a Linked Stream to Upload Large Files

If you have a large file to upload, consider using the **LinkToFile()** method of the stream interface. That is, instead of creating a stream and reading the file into it, create a stream and link it to the file. Use this linked stream when you call the **Store()** method of %Net.FtpSession.

For example:

Class Member

```
Method SendLargeFile(ftp As %Net.FtpSession, dir As %String, filename As %String)
{
    Set filestream=##class(%FileBinaryStream).%New()
    Set sc=filestream.LinkToFile(dir_filename)
    If $$$ISERR(sc) {do $System.Status.DisplayError(sc) quit }

    //Uploaded file will have same name as the original
    Set newname=filename

    Set sc=ftp.Store(newname,filestream)
    If $$$ISERR(sc) {do $System.Status.DisplayError(sc) quit }
}
```

4.4 Customizing Callbacks Issued by the FTP Server

You can customize the callbacks generated by the FTP server. By doing so, you can for example, give the user an indication that the server is still working on a large transfer, or allow the user to abort the transfer.

To customize the FTP callbacks:

1. Create a subclass of %Net.FtpCallback.
2. In this subclass, implement the **RetrieveCallback()** method, which is called at regular intervals when receiving data from the FTP server.
3. Also implement the **StoreCallback()** method, which is called at regular intervals when writing data to the FTP server.
4. When you create an FTP session (as described in "Establishing an FTP Session"), set the Callback property equal to your subclass of %Net.FtpCallback.

For details, see the class documentation for %Net.FtpCallback.

5

Sending and Receiving IBM WebSphere MQ Messages

Caché provides an interface to IBM WebSphere MQ, which you can use to exchange messages between Caché and the message queues of IBM WebSphere MQ. To use this interface, you must have access to an IBM WebSphere MQ server, and the IBM WebSphere MQ client must be running on the same machine as Caché.

The interface consists of the %Net.MQSend and %Net.MQRecv classes, which are both subclasses of %Net.abstractMQ. These classes use a dynamic-link library that is automatically installed by Caché on all suitable platforms. (This is MQInterface.dll on Windows; the file extension is different for other platforms.) In turn, the Caché dynamic-link library requires IBM WebSphere MQ dynamic-link libraries.

The interface supports sending and receiving only text data, not binary data. If you do not have long strings enabled, you can send and receive long messages by using file streams.

This chapter discusses the following topics:

To use IBM WebSphere MQ, you will need the formal documentation for this product. Also, for additional information on the Caché interface to IBM WebSphere MQ, see the class reference for %Net.abstractMQ.

5.1 Using the Caché Interface to IBM WebSphere MQ

In general, to use the Caché interface to IBM WebSphere MQ, you do the following:

1. Make sure that you have access to IBM WebSphere MQ v7.x or higher. Specifically:

- The IBM WebSphere MQ client must be installed on the same machine as Caché. Note that the installer updates the *PATH* environment variable and adds other system variables as needed.
- Make sure that you have rebooted the machine after installing the client, so that Caché is aware of the client.
- The client must have access to an IBM WebSphere MQ server.
- The username under which you will access the server must have permission to use the queue managers and the queues that you plan to use.

2. Create a new instance of %Net.MQSend or %Net.MQRecv, depending on whether you are going to send or receive messages.
3. Connect to an IBM WebSphere MQ server. When you do so, you provide the following information:
   - The name of a queue manager.
   - The name of a queue to use.
   - The channel by which to communicate with that queue. You specify a channel name, a transport mechanism, and the IP address and port of the IBM WebSphere MQ server.

   You can also provide a name and password if you are using the authentication feature of IBM WebSphere MQ.
4. Invoke the appropriate methods of %Net.MQSend or %Net.MQRecv to send or receive messages.

Note: To use IBM Websphere MQ on 64–bit Linux platforms, you must set the `LD_LIBRARY_PATH` to include the location of the MQ libraries. Because the path must be set for any Caché process that uses the MQ interface, it must be set prior to starting Caché if running background processes, and set in any UNIX® terminal prior to running csession.

5.1.1 Getting Error Codes

The methods of %Net.MQSend and %Net.MQRecv return either 1 if successful or 0 if unsuccessful. In the case of an error, call the **%GetLastError()** method, which returns the last reason code given by IBM WebSphere MQ. For information on the reason codes, see the formal IBM documentation.

5.2 Creating a Connection Object

Before you can send or receive messages via IBM WebSphere MQ, you must create a *connection object*, a Caché term for an object that establishes a connection to a queue manager, opens a channel, and opens a queue for use. There are two ways you can do this:

- You can use the %Init method, which takes arguments that specify all the needed information.
- You can use the %Connect method after first setting properties that specify all the needed information.

5.2.1 Using the %Init() Method

To use the %Init() method to create a connection object:

1. Create an instance of %Net.MQSend (if you are going to send messages) or %Net.MQRecv (if you are going to receive messages). This chapter refers to this instance as the connection object.

Note: If you receive the <DYNAMIC LIBRARY LOAD> error, a dynamic-link library is missing, and the cconsole.log file (in the system manager's directory) has more details.

2. Call the **%Init()** method of the connection object. This method takes the following arguments, in order.
   a. (Required) A string that specifies the queue name; this should be a valid queue for the specified queue manager.
   b. A string that specifies the queue manager; this should be a valid queue manager on the IBM WebSphere MQ server.

      If you omit this argument, the system uses the default queue manager, as configured in IBM WebSphere MQ. Or, if IBM WebSphere MQ has been the configured so that the queue manager is determined by the queue name, the system uses the queue manager that is appropriate for the given queue name.
   c. A string that specifies the specification for the channel, in the following form:

      ```
      "channel_name/transport/host_name(port)"
      ```

      Here *channel_name* is the name of the channel to use, *transport* is the transport used by the channel, *host_name* is the server name (or IP address) that is running the IBM WebSphere MQ server, and *port* is the port that this channel should use.

      Transport can be one of the following: `TCP`, `LU62`, `NETBIOS`, `SPX`

      For example:

      ```
      "CHAN_1/TCP/rodan(1401)"
      ```

      ```
      "CHAN_1/TCP/127.0.0.1(1401)"
      ```

      If you omit this argument, the system uses the default channel specification, as configured in IBM WebSphere MQ. Or, if the system has been the configured so that the channel is determined by the queue name, the system uses the channel that is appropriate for the given queue name.
   d. An optional string that specifies the log file to write error messages to. The default is that no logging occurs.

3. Check the value returned by the **%Init()** method. If the method returns 1, the connection was established successfully, and you can use the connection object to either send or receive messages (depending on the class you are using). See "Getting Error Codes."

5.2.2 Using the %Connect() Method

In some cases, you might prefer to specify all the details of the connection individually. To do so, you use the **%Connect()** method, as follows:

1. Create an instance of %Net.MQSend (if you are going to send messages) or %Net.MQRecv (if you are going to receive messages). As noted previously, this chapter refers to this instance as the connection object.

   Note: If you receive the <DYNAMIC LIBRARY LOAD> error, a dynamic-link library is missing, and the cconsole.log file (in the system manager's directory) has more details.

2. Set the following properties of the connection object:
   - QName — (Required) Specifies the queue name; this should be a valid queue for the specified queue manager.
   - QMgr — Specifies the queue manager to use; this should be a valid queue manager on the IBM WebSphere MQ server.

If you omit this argument, the system uses the default queue manager, as configured in IBM WebSphere MQ. Or, if IBM WebSphere MQ has been the configured so that the queue manager is determined by the queue name, the system uses the queue manager that is appropriate for the given queue name.

3. Optionally specify the channel to use by setting the following properties of the connection object:
   - Connection — Specifies the host and port of the IBM WebSphere MQ server. For example: `"127.0.0.1:1401"`.
   - Channel — Specifies the name of the channel to use. This must be a valid channel on the IBM WebSphere MQ server.
   - Transport — Specifies the transport used by the channel. This property can be one of the following: `"TCP"`, `"LU62"`, `"NETBIOS"`, `"SPX"`

   If you omit these arguments, the system uses the default channel specification, as configured in IBM WebSphere MQ. Or, if the system has been the configured so that the channel is determined by the queue name, the system uses the channel that is appropriate for the given queue name.
4. Call the **%ErrLog()** method of the connection object. This method takes one argument, the name of the log file to use for this connection object.
5. Check the value returned by the **%ErrLog()** method. See "Getting Error Codes."
6. Call the **%Connect()** method of the connection object.
7. Check the value returned by the **%Connect()** method. If the method returns 1, the connection was established successfully, and you can use the connection object to either send or receive messages (depending on the class you are using). See "Getting Error Codes."

5.3 Specifying the Character Set (CCSID)

To set the character set used for message conversions, call the **%SetCharSet()** method of your connection object. Specify an integer Coded Character Set ID (CCSID) as used in IBM WebSphere MQ.

- If you are sending messages, this should be the character set of those messages. If you do not specify a character set, the MQ system assumes the messages use the default character set specified for the MQ client.
- If you are retrieving messages, this is the character set to which those messages will be translated.

To get the CCSID that is currently being used, call the **%CharSet()** method. This method returns the CCSID by reference and it returns 1 or 0 to indicate whether it was successful; see "Getting Error Codes."

For information on the CCSID that corresponds to a given character set, see the formal IBM documentation.

5.4 Specifying Other Message Options

To specify message descriptor options, optionally set the following properties of your connection object:

- ApplIdentityData specifies the Application Identity message descriptor option.
- PutApplType specifies the Put Application Type message descriptor option.

5.5 Sending Messages

To send messages, do the following:

1. Create a connection object as described in "Creating a Connection Object." In this case, create an instance of %Net.MQSend. The connection object has a message queue to which you can send messages.
2. Call the following methods, as needed:
   - **%Put()** — Given a string, this method writes that string to the message queue.
   - **%PutStream()** — Given an initialized file character stream, this method writes that string to the message queue. Note that you must set the Filename property of the stream in order to initialize it. Binary streams are not supported.
   - **%SetMsgId()** — Given a string, this method uses that string as the message ID for the next message that is sent.
3. Check the value returned by the method you called. See "Getting Error Codes."
4. When you are done retrieving messages, call the **%Close()** method of the connection object to release the handle to the dynamic-link library.

5.5.1 Example 1: SendString()

The following class method sends a simple string message to the queue `cachetest`, using queue manager `QM_antigua`, and a queue channel named `S_antigua`. The channel uses TCP transport, and the IBM WebSphere MQ server is running on a machine called `antigua` and is listening on port 1401.

Class Member

```
///Method returns reason code from IBM WebSphere MQ
ClassMethod SendString() As %Integer
{
 Set send=##class(%Net.MQSend).%New()
 Set queue="cachetest"
 Set qm="QM_antigua"
 Set chan="S_antigua/TCP/antigua(1414)"
 Set logfile="c:\mq-send-log.txt"

 Set check=send.%Init(queue,qm,chan,logfile)
 If 'check  Quit send.%GetLastError()

 //send a unique message
 Set check=send.%Put("This is a test message "_$h)

 If 'check  Quit send.%GetLastError()
 Quit check
}
```

5.5.2 Example 2: SendCharacterStream()

The following class method sends the contents of a file character stream. It uses the same queue used in the previous example:

Class Member

```
///Method returns reason code from IBM WebSphere MQ
ClassMethod SendCharacterStream() As %Integer
{
 Set send=##class(%Net.MQSend).%New()
 Set queue="cachetest"
 Set qm="QM_antigua"
 Set chan="S_antigua/TCP/antigua(1414)"
 Set logfile="c:\mq-send-log.txt"
```

```
 Set check=send.%Init(queue,qm,chan,logfile)
 If 'check  Quit send.%GetLastError()

 //initialize the stream and tell it what file to use
 Set longmsg=##class(%FileCharacterStream).%New()
 Set longmsg.Filename="c:\input-sample.txt"

 Set check=send.%PutStream(longmsg)

 If 'check  Quit send.%GetLastError()
 Quit check
}
```

5.5.3 Example 3: Sending a Message from the Terminal

The following example shows a Terminal session that sends a message to an IBM WebSphere MQ queue. This works only on a machine that has been configured with the IBM WebSphere MQ client.

```
Set MySendQ = ##class(%Net.MQSend).%New()

Do MySendQ.%Init("Q_1", "QM_1","QC_1/TCP/127.0.0.1(1401)","C:\mq.log")

Do MySendQ.%Put("Hello from tester")

Set MyRecvQ =##class(%Net.MQRecv).%New()

Do MyRecvQ.%Init("Q_1", "QM_1","QC_1","C:\mq.log")

Do MyRecvQ.%Get(.msg, 10000)

Write msg,!
```

Also see the preceding sections for other examples.

5.6 Retrieving Messages

To retrieve messages, do the following:

1. Create a connection object as described in "Creating a Connection Object." In this case, create an instance of %Net.MQRecv. The connection object has a message queue from which you can retrieve messages.
2. Call the following methods, as needed:
   - **%Get()** — Returns a string message by reference as the first argument.
   - **%GetStream()** — Given an initialized file character stream, this method retrieves a message from the queue and places it into the file associated with that stream. Note that you must set the Filename property of the stream in order to initialize it. Binary streams are not supported.

   For both methods, the second argument is the timeout, in milliseconds; this controls the time used to contact the server. The default timeout is 0.
3. Check the value returned by the method you called. See "Getting Error Codes." Remember that IBM WebSphere MQ returns 2033 when the queue is empty.
4. When you are done retrieving messages, call the **%Close()** method of the connection object to release the handle to the dynamic-link library.

5.6.1 Example 1: ReceiveString()

The following class method retrieves a message from the `cachetest` queue.

Class Member

```
/// This is useful only if you enable long strings or you are
/// sure that you will receive strings < 32k in length.
///Method returns string or null or error message
ClassMethod ReceiveString() As %String
{
 Set recv=##class(%Net.MQRecv).%New()
 Set queue="cachetest"
 Set qm="QM_antigua"
 Set chan="S_antigua/TCP/antigua(1414)"
 Set logfile="c:\mq-recv-log.txt"

 Set check=recv.%Init(queue,qm,chan,logfile)
 If 'check  Quit recv.%GetLastError()

 Set check=recv.%Get(.msg)
 If 'check {
     Set reasoncode=recv.%GetLastError()
     If reasoncode=2033 Quit ""
     Quit "ERROR: "_reasoncode
     }

 Quit msg
}
```

5.6.2 Example 2: ReceiveCharacterStream()

The following method can retrieve a longer message because it uses **%GetStream()**:

Class Member

```
/// Method returns reason code from IBM WebSphere MQ
ClassMethod ReceiveCharacterStream() As %Integer
{
 Set recv=##class(%Net.MQRecv).%New()
 Set queue="cachetest"
 Set qm="QM_antigua"
 Set chan="S_antigua/TCP/antigua(1414)"
 Set logfile="c:\mq-recv-log.txt"

 Set check=recv.%Init(queue,qm,chan,logfile)
 If 'check  Quit recv.%GetLastError()

 //initialize the stream and tell it what file to use
 //make sure filename is unique we can tell what we received
 Set longmsg=##class(%FileCharacterStream).%New()
 Set longmsg.Filename="c:\mq-received"_$h_".txt"

 Set check=recv.%GetStream(longmsg)

 If 'check  Quit recv.%GetLastError()
 Quit check
}
```

5.7 Updating Message Information

The %Net.MQSend and %Net.MQRecv classes also provide the following methods:

%CorId()

Updates (by reference) the Correlation Id for the last message read.

%ReplyQMgrName()

Updates (by reference) the reply queue manager name for the last message read.

%ReplyQName()

Updates (by reference) the reply queue name for the last message read.

5.8 Troubleshooting

If you encounter problems when using the Caché interface to IBM WebSphere MQ, you should first determine whether the client is correctly installed and can communicate with the server. To perform such a test, you can use sample programs that are provided by IBM WebSphere MQ. The executables are in the bin directory of the IBM WebSphere MQ client.

The following steps describe how to use these sample programs on Windows. The details may be different on other operating systems; consult the IBM documentation and check the names of the files present in your client.

1. Create an environment variable called *MQSERVER*. Its value should be of the form *channel_name/transport/server*, where *channel_name* is the name of the channel to use, *transport* is a string that indicates the transport to use, and *server* is the name of the server. For example: `S_antigua/TCP/antigua`
2. At the command line, enter the following command:

   ```
   amqsputc queue_name queue_manager_name
   ```

   where *queue_name* is the name of the queue to use and *queue_manager_name* is the name of the queue manager. For example:

   ```
   amqsputc cachetest QM_antigua
   ```

   If the `amqsputc` command is unrecognized, make sure that the *PATH* environment variable has been updated to include the bin directory of the IBM WebSphere MQ client.

   In case of other errors, consult the IBM documentation.
3. You should see a couple of lines like the following:

   ```
   Sample AMQSPUT0 start
   target queue is cachetest
   ```

4. Now you can send messages. Simply type each message and press Enter after each message. For example:

   ```
   sample message 1
   sample message 2
   ```

5. When you are done sending messages, press Enter twice. You will then see a line like the following:

   ```
   Sample AMQSPUT0 end
   ```

6. To complete this test, we will retrieve the messages you sent to the queue. Type the following command at the command line:

   ```
   amqsgetc queue_name queue_manager_name
   ```

   where *queue_name* is the name of the queue to use and *queue_manager_name* is the name of the queue manager. For example:
7. You should then see a start line, followed by the messages that you sent previously, as follows:

   ```
   Sample AMQSGET0 start
   message <sample message 1>
   message <sample message 2>
   ```

8. This sample program waits briefly to receive any other messages and then displays the following:

```
no more messages
Sample AMQSGET0 end
```

If the test fails, consult the IBM documentation. Possible causes of problems include the following:

- Security issues
- Queue is not defined correctly
- Queue manager is not started

6

Using SSH

The %Net.SSH package provides support for SSH (Secure Shell) communications. This chapter briefly introduces the classes in this package.

6.1 Creating an SSH Session

%Net.SSH.Session represents an SSH session. To use this class:

1. Create an instance of the class.
2. Use the **Connect()** instance method to connect to a server.
3. Use either **AuthenticateWithKeyPair()** or **AuthenticateWithUsername()** to authenticate yourself to the server. For details, see the class reference for %Net.SSH.Session.
4. Use additional methods of %Net.SSH.Session to perform SCP (Secure Copy) operations of single files to and from the remote system, execute remote commands, tunnel TCP traffic, or perform SFTP operations. See the class reference for %Net.SSH.Session.

   For example, use **OpenSFTP** to use the session for SFTP operations. This method returns, by reference, an instance of %Net.SSH.SFTP that you can use for SFTP operations. See the example in the next section.

Important: For information on the supported platforms where you can use these classes, see the class reference for %Net.SSH.Session and %Net.SSH.SFTP.

6.2 Example: Listing Files via SFTP

The following method shows how you can write a list of the files on a server, via SFTP:

Class Member

```
Method SFTPDir(ftpserver, username, password) As %Status
{
    set ssh = ##class(%Net.SSH.Session).%New()
    set status = ssh.Connect(ftpserver)
    set status = ssh.AuthenticateWithUsername(username,password)
    //open an SFTP session and get that returned by reference
    set status = ssh.OpenSFTP(.sftp)
    //get a list of files
    set status = sftp.Dir(".",.files)
    set i=$ORDER(files(""))
    while i'="" {
        write $listget(files(i),1),!
        set i=$ORDER(files(i))
    }
    quit $$$OK
}
```

6.3 Additional Examples

For additional SSH examples, open %Net.SSH.Session in Studio and see the **TestExecute()** and **TestForwardPort()** methods of this class.

7

Other Caché %Net Tools

Here is a brief list of some other useful classes in %Net:

%Net.URLParser

Caché provides a utility class, %Net.URLParser, that you can use to parse URL strings into their component parts. This is useful, for example, when you are redirecting an HTTP request.

This class contains one class method, **Parse()**, that takes a string containing a URL value and returns, by reference, an array that contains the parts of the URL. For example:

```
Set url = "https://www.intersys.com/main.csp?QUERY=abc#anchor"
Do ##class(%Net.URLParser).Parse(url,.components)
```

Upon return, *components* will contain an array of the parts of this URL:

Element	Value	Description
components(“scheme”)	http:	The transport scheme specified by this URL.
components(“netloc”)	www.intersys.com	The network address of the URL.
components(“path”)	/main.csp	The file path of the URL.
components(“query”)	QUERY=abc	The query string associated with the URL.
components(“fragment”)	anchor	The fragment (following the # character) for the URL.

For more information, refer to the class documentation for %Net.URLParser.

%Net.Charset

You can use %Net.Charset to represent MIME character sets within Caché and map these character sets to Caché locales. This class provides the following class methods:

- **GetDefaultCharset()** returns the default character set for the current Caché locale,
- **GetTranslateTable()** returns the name of the Caché translation table for a given input character set.
- **IsLocaleUnicode()** indicates whether the current locale is Unicode.
- **TranslateTableExists()** indicates whether the translation table for the given character set has been loaded.

For method signatures, see the class documentation for %Net.Charset.

For more information on character sets and translation tables, see "System Classes for National Language Support" in *Caché Specialized System Tools and Utilities*.

%Net.TelnetStream

You can use %Net.TelnetStream to emulate the handshaking behavior of Windows NT Telnet.exe. For details, see the class documentation for %Net.TelnetStream.

%Net Security Classes

The %Net package provides many classes for authentication and security. For information, see the extensive class documentation.