Conga 3.1 User Guide – Supplement

Version dated 18 July 2018.

Introduction

The version of Conga included with Dyalog version 17.0 has the version number 3.1. The only new functionality in version 3.1 compared to 3.0 is support for a different format for decimal floating-point numbers used by version 17.0. This *Conga 3.1 User Guide Supplement* is a slightly enhanced version of the supplement that was released with Conga 3.0 and Dyalog APL version 16.0. It contrasts version 3.0 to earlier versions and provides guidance on how to take advantage of the significant new features of versions 3.x when compared to 2.x.

Version number 3.0 will be used in the remainder of this document: everything in this document applies equally to versions 3.0 and 3.1; the new document provides some additional detail on features introduced with Conga version 3.0. The material found this supplement will be merged into a forthcoming version of the *Conga User Guide*. Revisions of this document and the *Conga User Guide* will be released through the Documentation Centre as they become available.

Using the Conga User Guide

With the exception of some sample applications described in the document, which have been moved from the distributed workspace **conga.dws** to new locations in Dyalog releases 16.0 and 17.0, the *Conga User Guide* can be used with any recent version of Conga, 2.7, 3.0 or 3.1: Although it allows new usage patterns, Conga version 3.0 is designed to be upwardly compatible with Conga version 2.7 and earlier releases.

If your application requires any of the sample components that have been (re)moved, refer to the section titled Code Compatibility and Distribution for instructions on how to download **conga_v2.dws**, which is provided as a temporary bridging solution.

New Features of Version 3.0

Conga version 3.0 provides many new features designed to make it easier to write network-based applications:

1. **Multiple Isolated "Roots"** make it possible for separate components running in the same Dyalog process to use Conga without interfering with each other in any way. This allows development tools to use TCP connections without interfering with the application they are being used to develop.

2. **New Server modes** efficiently support different server styles or requirements:
   - *FIFOMode*: Process incoming messages in strict chronological order (does not allow selection of messages arriving on individual connections)
   - *ConnectionOnly*: The server object will only receive connection events; the application is expected to spawn individual APL threads to listen on each connection object separately.

3. ***Timeout* and *Disconnect* as events** instead of error codes 100 and 1119. Timeouts and disconnections can optionally be signalled as events to simplify server logic.

4. **Temporarily prevent new connections** by setting the *Pause* property.

5. ***Sent* event** allows programs transferring large amounts of data to avoid flooding buffers with untransmitted data by requesting a *Sent* event when the last byte has been transmitted, and using this to trigger transfer of the next data block.

6. **Transmit files** without first reading their data into the APL workspace.

7. **HTTP protocol support** means that, rather than receiving text blocks and deciphering them, you can elect to receive *HTTPHeader*, *HTTPBody*, *HTTPChunk*, *HTTPTrailer* events after Conga has parsed the incoming messages. This greatly simplifies the handling of HTTP messages.

8. **WebSocket support** means that HTTP connections can be upgraded to bi-directional web sockets, allowing asynchronous bidirectional data transmission for highly interactive user interfaces.

9. **Allow or Deny connections from specific address ranges** to support applications that only want to service connections made from certain locations. Provides simple protection against denial or service or other malicious attacks.

10. **Conga 3.1 upgraded to support for GnuTLS 3.5.16** to provide secure communications. Conga 3.0 previously upgraded to support for GnuTLS 3.4.16.

11. **Dynamic loading of secure socket support** makes it unnecessary to install or ship the secure part of Conga with your application if you do not intend to make use of secure features.

12. **Shared Unicode/Classic library** means that the same **.so** or **.dll** file is used by Classic and Unicode versions of Conga (32 and 64 bit are still separate).

13. **Simpler configuration** – by default, Conga libraries are always loaded from the folder where the interpreter executable is located, bypassing the need for LIBPATH or similar environment variables.

14. **A numeric *Version* property** makes it easier to write applications that need to know which version of Conga they have available.

15. **Experimental UDP support:** Undocumented; contact support@dyalog.com for more information.

16. **Numerous new samples and tools**.

Compatibility and Code Distribution

Conga 3.0 is designed to be compatible with Conga 2.x, but the APL code that wraps the Conga libraries has been substantially rewritten to provide support for multiple isolated roots and to provide new, simplified ways to create client and server applications. At the same time, a number of samples have been moved from the distributed **conga.dws** to other locations – and a small number have been removed completely.

- The TODServer example has been completely retired.
- The FTPClient, along with many of the utilities that used to be found in the HttpUtils and Samples namespaces, have been moved to new locations (see the *Code Libraries Reference Guide*).
- The RPCServer and WebServer examples are replaced by new code that uses new features of Conga 3.0. These can be found in the **[DYALOG]/Samples/Conga** directory and are described in this document.

If your application requires any of the components that have been (re)moved, then a workspace called **conga_v2.dws** is available from the Conga section of the Tools Download page on https://my.dyalog.com. The **conga_v2.dws** workspace contains all the old code but loads the Conga 3.1 DLLs that are provided with Dyalog version 17.0. If you do decide to use the **conga_v2.dws** workspace, Dyalog asks that you notify support@dyalog.com and inform us why you felt this was necessary so that we can consider reinstating code and improve future releases.

Initialisation

To take advantage of some of the new features of Conga version 3.0, you will need to make changes to the way that you initialise Conga-based applications.

Existing Conga 2.x-based applications typically initialise Conga by calling (`DRC.Init ''`). Unless you want to make use of the multiple-root feature described later in this document, we recommend creating a default instance of Conga 3 as follows:

```
      iConga←Conga.Init ''
```

The above expression connects to the Conga root object named DEFAULT, which is shared by all applications that use an empty right argument to `Init`. If you name the instance variable DRC, existing application code that refers to DRC (except for the call to `DRC.Init`) should subsequently work unchanged. In other words, if you replace the expression (`DRC.Init ''`) with (`DRC←Conga.Init ''`) and ensure that you have the Conga namespace rather than DRC loaded into your workspace, the rest of your Conga 2.x application should run unchanged.

Many of the code samples that used to reside in the `Samples` namespace in earlier Conga workspaces have been withdrawn as better alternatives now exist. A cut-down set of the most widely used samples can be found in the `Samples` namespace in **conga.dws**. A new set of samples can be found in the **Samples/Conga** directory, and tools such as the new HttpCommand which is intended to replace the old `Samples.HttpGet` function, can be found in the **Library/Conga** directory.

New Features in Conga 3.0

1. Multiple Isolated Roots

As the use of Conga has grown, it has become common for more than one component in an application to use Conga, leading to potential name conflicts between client and server objects created, and clashes between different state settings. In particular, when an application and the tools used to maintain it both use TCP communications, they need to be independent of each other. Each may need to be restarted or reset without interfering with the other. Conga version 3.0 allows each component to have its own "Root" under which Conga objects are created and perform operations (like deleting all existing Conga connections in order to restart) without fear of interference with other components.

Prior to Conga version 3.0, all components in the same process used the same DRC namespace: the first user would call `DRC.Init` and received a clean return code, and subsequent calls to `Init` warned that Conga was already initialised. If a component decided to reset or re-initialise Conga, all other components would be affected by this.

With Conga version 3.0, the DRC namespace is still provided for backwards compatibility, but – as mentioned in the previous section - it is recommended that you use the `Conga.Init` function to create or select a specific named root for your application or component rather than calling `DRC.Init`. For ad hoc use, you can use an empty right argument to connect to the default root, but for an application that might need to manage the state of Conga, Dyalog recommends using a right argument to identify your application.

```
      iConga←Conga.Init 'MyApp'
```

The above statement can safely be called anywhere in your application; if a root with that name already exists, then a reference will be returned to the existing root instance. If you want to be sure that a new instance is created, you can use `Conga.New`. In this case, an empty argument will generate a new unused root name, and a non-empty argument will signal an error if the root name is already in use. The function `RootNames` can be used to get a complete list of existing roots:

```
      iC1←Conga.Init '' ⍝ Use the default root
      iC2←Conga.New ''  ⍝ Create a new one with a generated name
      Conga.RootNames
 DEFAULT  IC1
```

To re-initialise your root, erase the reference (all references) to the instance; it will be cleaned up, and you can create a new one.

Warning: The intention is not that you create a large number of roots. The process of creating and tearing down roots is expensive and complex. Components might need a separate root, but you should not create new roots just to (for example) create queries on the internet – in this situation, use the default root that you can get a reference to by passing an empty argument to `Conga.Init`.

2. New Server Modes

The default mode for a server allows an application to selectively wait on the entire Server, receiving both connection-related events and data transmissions to all or part of the tree of objects that make up the server. As usage patterns evolve, Conga is also evolving to provide modes that are better suited to, or tuned for, these patterns.

2.a FIFOMode

For high volume services with hundreds or thousands of connections, the cost of providing filtering functionality becomes unacceptable. In addition, the filtering mechanism can lead to some connections receiving better service than others. The `FIFOMode` switch turns off the ability to call the `Wait` function on a subset of a Server object hierarchy. In return, you get significantly less CPU consumption, and are guaranteed that messages come off the queue in strict chronological order of arrival.

Enable *FIFOMode* for a server using `SetProp`:

```
      iConga.SetProp 'S1' 'FIFOMode' 1
```

With *FIFOMode* enabled, attempts to Wait on a connection object that is a child of the server will fail with error `1142 ERR_FIFOMODE`.

2.b ConnectionOnly

Some server applications have a structure that makes it convenient to launch an APL thread for each client connection and leave that thread running for the duration of that client session. The *ConnectionOnly* switch enables a mode where `Wait` on the server object will only ever report `Connect` events; individual application threads are expected to call `Wait` on the connection that they are managing.

Samples/RPCServices/ThreadedRPC contains an example of a server that uses ConnectionOnly and runs a thread for each connection.

3. Timeout and Close as Events

Many application developers have found it inconvenient that "normal" events such as a timeout due to inactivity or the closing of a connection are reported as if they were errors with non-zero return codes from `Wait`, rather than being classified as events.

In Conga version 3.0 it is possible to receive timeout and close as events. However, as this is a **breaking change**, it is not enabled by default and you need to explicitly set the *EventMode* property to enable it. The property is set on the root object and applies to all clients and servers created as children of that root.

Dyalog strongly recommends that you enable *EventMode,* and it is likely that this will become the default in a future version of Conga.

To enable Event Mode, use:

```
      iConga.SetProp '.' 'EventMode' 1
```

Once EventMode is set, return codes 100 and 1119 will no longer issued. The different formats for the result from `Wait` are:

`'EventMode' 0`	`'EventMode' 1`
`100  'TIMEOUT' ''`	`0 'S1' 'Timeout' 100`
`1119 'CLOSE'   ''`	`0 'S1' 'Close'   1119`

4. Temporarily Prevent New Connections to a Server

If a server needs a break from incoming connections (for example, because it is preparing to shut down for maintenance or is overloaded), this can be achieved the *Pause* property. The *Pause* property has three possible settings:

Setting	Effect
1	Keeps the listening socket open but does not accept new incoming connections. Connection attempts that have not timed out on the client side will be accepted when Pause is set to 0.
2	Closes the socket but keeps the server object alive. When Pause is set to 0 the socket will be re-created.
0	Resume normal operations.

For example:

```
      iConga.SetProp 'S1' 'Pause' 1 ⍝ Do not accept connections
```

5. Sent event

If you are transmitting a large amount of data in chunks, Conga allows you to make repeated calls to the `Send` function without waiting for the previous send to complete. This can cause large amounts of data to accumulate in buffers either in Conga or the network layer, which might be undesirable – it also makes it difficult to cancel an operation, since a large number of operations are already queued.

In Conga version 3.0, you can request a receipt upon the completion of the actual transmission by appending a 3 following the data passed to the `Send` function:

```
      iConga.Send 'C1' data 3
0
      iConga.Wait 'C1'
0  C1  Sent  0
```

When using *Command* mode, the `Sent` event will be overridden by the answer on a command; if the response to a command arrives before you enter the next `Wait`, you will simply get the response and the `Sent` event will be suppressed.

6. File Transmission

When an APL-based server needs to transmit the entire contents of a file, earlier versions of Conga required that you first read the contents of the file into the APL workspace and then pass it to the `Send` function – an inefficient process.

`Send` now accepts a two-element nested vector; the first element is data to be transmitted first and the second element contains a file name, the contents of which will be transmitted after the initial data. The first element allows you to prepend information to the transmission, where necessary. For example:

```
      iConga.Send 'C1' ('' 'c:\mywebsite\index.html')
0
```

The file transmission mechanism does not apply to Command mode connections, in which each transmissions is an APL array which is transmitted in binary format.

7. HTTP Protocol Support

A common use of Conga is to act as an HTTP client (retrieving data from web sites or making web service requests) or as an HTTP server (serving up data managed by an APL application). In previous Conga versions this required buffering data and parsing the HTTP protocol in APL.

Beginning with Conga version 3.0, you can set the mode of any Client or Server to be *HTTP*. If you do this, the normal Receive and Block events are replaced with events that signal the arrival of a complete piece of HTTP protocol: HTTPHeader, HTTPBody, HTTPChunk and HTTPTrailer. This not only simplifies the task of receiving HTTP data, it also significantly improves performance by moving the parsing into multi-threaded, asynchronous C code dedicated to this task.

By default, Conga does not provide any further processing of received HTTP messages (for example, taking headers apart or decoding base-64 encoded data). Setting the *DecodeBuffers* property on the client or server will cause Conga to parse the pieces of the HTTP message and return them in a more convenient format.

When transmitting data, you are required to generate valid HTTP messages. The only support that Conga version 3.0 provides is to add a valid Content-Length header when transmitting a file (see File Transmission).

The `HttpUtils` namespace, which can be loaded using `]Load HttpUtils`, provides code that can be used to work with HTTP requests and responses. This is located in the **Library/Conga** directory and described in the *Code Libraries Reference Guide*.

Additional details on HTTP protocol support are provided later in this document.

8. Web Sockets

An established HTTP connection can be upgraded to bi-directional websocket connection; this allows both client and server to transmit data at any time rather than sticking to the normal cycle of the client making a request followed by a server response.

Web Socket Upgrade – Client Side

It is the client that requests the upgrade. To upgrade a Conga-based client, set the *WSFeatures* property to whether you want to automatically accept a positive response from the server (1) or you need to validate the response and confirm it (0). Unless you are familiar with websocket internals, auto-upgrade is recommended.

```
      iConga.SetProp clt 'WSFeatures' 0 ⍝ Do not automatically accept
0
```

Next, set the *WSUpgrade* property to a three-element vector containing a URL, hostname (normally the same one you already connected to) and any necessary header information that the particular server you are connecting to may be looking for in order to decide how to handle the connection. For example:

```
      iConga.SetProp 'C1' 'WSUpgrade' ('/' 'localhost' 'some-setting: value')
0
```

Now call `Wait` – if the server accepts your request the response will be a *WSResponse* event, with data containing header information that the server has decided to send:

```
      res
0  C1  WSResponse  HTTP/1.1 101 Switching Protocols
                   Upgrade: websocket
                   Connection: Upgrade
                   Sec-Websocket-Accept: G/cEt4HtsYEnPOMnSVkKRk459gM=
```

If *WSFeatures* had been set to 1, the response would have been a WSUpgrade event:

```
0  C1  WSUpgrade  0
```

If auto-accept is not enabled, we now need to examine the headers, decide whether they are OK, and finally set the *WSAccept* property:

```
      iConga.SetProp 'C1' 'WSAccept' ((4⊃res)'')
0
```

You are required to confirm the headers that you want to accept as the first element. The second element is not used but required for symmetry with the server call (see the next section). The next call to `Wait` should return a WSResponse event, after which the socket can be used as a websocket.

```
0  C1  WSResponse  HTTP/1.1 101 Switching Protocols
                   Upgrade: websocket
                   Connection: Upgrade
                   Sec-Websocket-Accept: G/cEt4HtsYEnPOMnSVkKRk459gM=
```

Web Socket Upgrade – Server Side

When a client requests a web socket upgrade, the server will either receive a *WSUpgrade* (if it has WSFeatures set to 1) or a *WSUpgradeReq* event from a call to `Wait`:

```
0  S1.CON00000000  WSUpgradeReq  GET / HTTP/1.1
                                 Host: localhost
                                 Upgrade: websocket
                                 Connection: Upgrade
                                 some-setting: value
                                 Sec-WebSocket-Version: 13
                                 Sec-WebSocket-Key: KSO+hOFs1q5SkEnx8bvp6w==
```

The format of the messages is the same; in the event of a *WSUpgrade* the 4th element is for information only and the upgrade has been done, otherwise you need to follow a similar pattern as the client, either closing the connection if the request is denied or responding with a confirmation of the received headers plus any information that need to be sent to the client:

```
      iConga.SetProp 'S1.CON00000000' 'WSAccept' ((4⊃res)'server-says: hello')
0
```

At this point, even before the client has received a *WSResponse* (if it is a Conga client, or the equivalent in JavaScript or some other programming language), the socket is considered open and the server should be able to transmit data. However, it may be prudent to wait for the client to initiate communications, as final confirmation that the websocket is operational.

Transmitting Data on a Web Socket

Once an HTTP connection has been upgraded to a Web Socket, the `Send` function can be used to transmit data. The specification of the data to send must be a 2 or 3 element vector, where the first element is the data to be transmitted as a character or integer vector and the second element is a Boolean that declares whether this is the final transmission in a sequence. An optional 3rd element specifies the "operation code"; the opcode can be 1 for Text (data must be character and will be converted to UTF-8), 2 for Binary (values between ¯128 and 127) or 0 for a "continuation", in which case the data must have the same type as your earlier transmission. For example:

```
      iConga.Send 'C1' ('Hello there' 1 1)
0
```

Incoming data is returned by `iConga.Wait` in the form of a WSReceive event, which returns the same data elements as the argument to `Send` described above: (`data final opcode`).

9. Allow or Deny Connections from Specific Address Ranges

Sets of IPv4 and IPv6 addresses that connections will be allowed from can be specified – or conversely ranges that will be denied. This means that you do not need to perform validation of valid peer addresses in APL application code.

Address ranges are set by setting the *AllowEndPoints* or *DenyEndPoints* properties when a server is started. If both are specified, then the intersections between the ranges will be disallowed. Each set of ranges is specified in the format:

```
┌→──────────────────────────────────────────────────────────────────────────────────────┐
│ ┌→───────┐ ┌→──────────────────────────────────────────────────────────────────────┐ │
│ │property│ │ ┌→────────────────────────────────────────────────┐ ┌→──────────────┐ │ │
│ └────────┘ │ │ ┌→───┐ ┌→──────────────────────────────────────┐ │ │ ┌→───┐ ┌→───────────────┐ │ │ │
│            │ │ │IPv6│ │fe80::d189:fd4:7003:a0a3/120,fe80::9df3:f956:84f5:12ab/120│ │ │ │IPv4│ │192.168.202.1/24│ │ │ │
│            │ │ └────┘ └──────────────────────────────────────┘ │ │ └────┘ └────────────────┘ │ │ │
│            │ └∊────────────────────────────────────────────────┘ └∊──────────────┘ │ │
│            └∊──────────────────────────────────────────────────────────────────────┘ │
└∊──────────────────────────────────────────────────────────────────────────────────────┘
```

Where "property" is replaced by either AllowEndPoints or DenyEndPoints. You can specify IPv4 and/or IPV6 sections and have any number of ranges in each section.

For example:

```
      allow←,⊂'IPV4' '192.168.1.1/127,10.17.221.67/75'
      iConga.Srv '' 'localhost' 8080 ('AllowEndPoints' allow)
```

Will cause the server to accept connections only from IP addresses 192.168.1.1 through 192.168.1.127 and 10.17.221.67 through 10.17.221.75.

10. Support for GnuTLS 3.5.16

In Conga 3.1, secure socket support is built upon GnuTLS version 3.5.16 (3.0 uses 3.4.16). Compared to Conga 2.7, the internal details of how certificates are handed have changed significantly, but there should be no user-visible changes except that secure connections are more likely to succeed and that it is possible to use certificates that are stored in the Microsoft Certificate store for Servers as well as clients. Conga version 3.0 connections will provide Server Name Indication and Session Tickets – but none of this is visible from the APL application.

11. Dynamic Loading of Secure Socket Libraries

The secure socket libraries, with names beginning with conga*nn*ssl, are loaded on demand when the first secure connection is created. If you do not include these libraries when installing applications that use Conga, you will be able to use all non-secure features of Conga. Any attempt to use secure features will fail.

12. Shared Unicode/Classic Library

The same **.dll** or **.so** library is now used by Classic and Unicode editions of Dyalog APL. This should have no user-visible effects other than that the file name is the same for both editions.

13. Simple Configuration

By default, Conga libraries are always loaded from the directory where the interpreter executable is located, bypassing the need for LIBPATH or similar environment variables. This simplifies the installation of Conga-based applications, since Conga will work if all the required DLLs are placed in the same folder as the interpreter without requiring further configuration of the system.

13. Numeric Version Property

`Version` now returns a 3-element integer vector containing the major and minor version numbers, followed by the SVN revision number of the source code used to build the current version. For example:

```
      iConga.Version
3 1 1405
```

14. Experimental UDP Support

UDP support is available but is still being prototyped in collaboration with potential users. Please contact support@dyalog.com for more information.

15. Numerous New Samples

Several new classes and namespaces are provided as examples of how to construct different types of services, as well as utilities to make it easier to make client requests of HTTP and other services. The **[DYALOG]/Library/Conga** directory contains FtpClient, HttpCommand and HttpUtils; these are described in the *Code Libraries Reference Guide*.

The **Samples/Conga** directory is cloned from the GitHub repository https://github.com/Dyalog/samples-conga. The GitHub repository, and the documentation that it contains, are likely to be updated as features are added and corrections made. We recommend that you visit it periodically. At the time of Dyalog version 17.0's release it had three sub-folders which provide code samples or utilities that might be useful in providing or consuming Conga-based services.

The GitHub repository https://github.com/Dyalog/conga-apl contains the source code for the `conga` workspace, and within it there is a directory named **Tests**. The **Tests** directory contains the test scripts used by Dyalog Ltd to test Conga version 3.1. These tests also provide working examples of old and new Conga features. Contributions are very welcome!

Directory CertTool

This directory currently contains a single sample that shows how the GnuTLS certTool can be used to generate self-signed certificates.

Directory HttpServers

This directory contains several HTTP server classes which can be use as arguments to the new `Conga.Srv` function. Most of the samples can be loaded and run using a recipe similar to the following:

```
    ]Load Samples/Conga/DocHttpRequest
    iSrv←Conga.Srv 8088 DocHttpRequest
    iSrv.Start

    ]Load HttpCommand
    (rc headers data)←HttpCommand.Get 'http://localhost:8088/index.html'
```

Directory RPCServices

This directory folder contains examples of simple servers that can be used to make remote procedure calls using Conga's Command mode, in which each transmission consists of an APL array.

HTTP Protocol Support

Enabling the HTTP Protocol

The HTTP protocol can be enabled specifying `'http'` as the mode parameter when then client or server is created.

```
      client ← iConga.Clt 'C1' 'www.dyalog.com' 80 'http'
      server ← iConga.Srv 'S1' 'localhost' 8080 'http'
```

Receiving HTTP Messages

Whether acting as a client or server, the process of receiving HTTP messages is the same and will follow one of three patterns.

1) HTTP Header only. The entire message is contained in the message header. This is typical with the HTTP GET method – all of the information is passed in the HTTP header and there is no body. This is indicated by a 0 (zero) value in the Content-Length HTTP header field.

2) HTTP Header and Body. The message is split between the HTTP header block and the HTTP body block. The body contains the data or payload for the message. The size of the body is indicated by the Content-Length header field.

3) HTTP Header, one or more Chunks, followed by a Trailer. When the payload is too large, or is being generated spontaneously, the payload can be split and transmitted in several chunks. A trailer, possibly empty, is sent after the last chunk. Chunked mode is indicated by the Transfer-Encoding header field having the value "chunked" and there being no Content-Length header field.

Conga version 3.0's HTTP mode introduces four events, one for each of the types of message above. The events are HTTPHeader, HTTPBody, HTTPChunk and HTTPTrailer. When any of these events occurs, the format of the event's data element depends on the setting of the server/client/connection object's *DecodeBuffers* property. Each HTTP event has a related, additive, *DecodeBuffers* value. When *DecodeBuffers* is not set for an HTTP event, Conga will return a simple character vector that you will then need to parse on your own. When *DecodeBuffers* is set, Conga will do the parsing and return an array of elements that are meaningful to the particular event.

Typically, you will set *DecodeBuffers* to 15 to decode the data for all HTTP-related events.

```
      iConga.SetProp obj 'DecodeBuffers' 15
```

where **`obj`** is the connection, server, or client object.

Event/Value		Data format without DecodeBuffer set	Data format with DecodeBuffer set

HTTPHeader	1	FirstLine, CRLF, Headers, CRLF,CRLF
		For a request, the first line is an HTTP command, a URI and HTTP version:
		`GET /index.html HTTP/1.1`
		For a response, the first line is HTTP version, HTTP status code and HTTP status text:
		`HTTP/1.1 200 OK`
		Subsequent lines have HTTP headers
		`Host: www.someplace.com`
		`User-Agent: Dyalog/Conga`
		`Accept: */*`
		`Accept-Encoding: gzip, deflate`
		A Content-Length header with a value of 0 indicates the message consists of a header only and you have received the entire message.
		
		If there is a Transfer-Encoding header with a value of 'chunked', the message will be sent in chunked format and you will need to loop and receive some number of chunks and ultimately a trailer.
			A 4-element array whose first 3 elements vary based on whether the message is a request or response.
			[1] Method (request) or HTTP version (response)
			[2] URI (request) or HTTP status code (response)
			[3] HTTP version (request) or HTTP status (response)
			[4] 2-column matrix of header name/value pairs
			[;1] header name
			[;2] header value
			
			Note: your application may still need to do further decoding of the header values (e.g. some values may be Base64 encoded)

HTTPTrailer	2	'0', CRLF, Trailers, CRLF Character vector (possibly empty) of lines separated by CRLF. Each line contains an HTTP header. Upon receipt of the trailer, you have received the entire message.	A 2-column matrix of header name/value pairs [;1] header name [;2] header value Note: your application may still need to do further decoding of the header values (e.g. some values may be Base64 encoded)
HTTPChunk	4	HexChunkLength, [;Chunk-Extension(s)], CRLF, Chunk-Text, CRLF	A 2-element array of [1] character vector chunk message text [2] 2-column matrix of chunk-extension name/value pairs [;1] chunk-extension name [;2] chunk-extension value
HTTPBody	8	Character vector of the message body	Character vector of the message body

Sending HTTP Messages

The HTTP-related event types are used to receive HTTP messages. To send an HTTP message you can either

- Compose a properly formatted HTTP message as a character vector and pass it to `Send`
- Pass `Send` a 5-element array. The contents of the first three elements depend on whether the message is a request or a response.
  [1] HTTP method (request) or HTTP version (response)
  [2] URI (request) or HTTP status code (response)
  [3] HTTP version (request) or HTTP status test (response)
  [4] 2-column matrix of header name/value pairs

[5] character vector message body
or a 2-element vector of `('' 'filename')` when sending the contents of a file

HTTP Utility Libraries

As noted earlier in this document, Conga itself does no further processing of the message data; any additional processing is the responsibility of the user. To address this situation, Dyalog has provided two utility libraries, `HttpCommand` and `HttpUtils`, in the **[DYALOG]/Library/Conga/** directory; these can be loaded using the SALT `Load` function. Both of the following statements will load `HttpUtils`, though the latter is suitable for running under program control. To load `HttpCommand`, substitute `HttpCommand` for `HttpUtils` in the statements below.

```
]Load HttpUtils
⎕SE.SALT.Load 'HttpUtils'
```

`HttpUtils` and `HttpCommand` are maintained in the GitHub repository found at https://github.com/Dyalog/library-conga. If you access the GitHub repository rather than simply using the versions installed with Dyalog APL, you will be able to see the revision history, download the latest versions and participate in the development by reporting issues and by posting questions and suggestions.

`HttpCommand` is a stand-alone utility to act at an HTTP client, sending HTTP messages and receiving responses. It can be used to interact with web services, retrieve web pages and other information from the internet. `HttpCommand` is documented in the *Code Libraries Reference Guide*.

`HttpUtils` is a namespace containing classes and utility functions to manipulate HTTP messages. `HttpUtils` was conceived and developed specifically in support of Conga 3.0's HTTP mode and the HTMLRenderer GUI object introduced in Dyalog version 16.0. As `HttpUtils` is a very new library, we welcome suggestions for improvements and additional features.

`HttpUtils` has two primary classes, `HttpRequest` and `HttpResponse`. These can both be used by clients and servers.

When acting as a client:

1) use `HttpRequest` to build and format the request to be sent to a server.
2) use `HttpResponse` to collect and convert the response data into an easier-to-use format.

Conversely, when acting as a server:

1) use `HttpRequest` to collect incoming requests from clients.
2) use `HttpResponse` to build and format the responses to be sent back to the clients.

The techniques for sending requests and receiving responses or receiving requests and sending responses are very similar and follow the pattern in the following code.

```
clt←DRC.Clt '' address port 'HTTP'  ⍝ start a client in HTTP mode
req←⎕NEW #.HttpUtils.HttpRequest    ⍝ create a new request
req.(Command Uri)←'GET' 'http://some-address'  ⍝ set some fields
:If 0=⊃DRC.Send clt req.Format      ⍝ format and send the request
  :Repeat
    :If ~done←0≠err←1⊃rc←DRC.Wait clt 5000  ⍝ standard Conga loop
      (err obj evt dat)←4↑rc  ⍝ break out the results

      :Select evt   ⍝  which Conga event?
      :Case 'Connect'
        resp←⎕NEW #.HttpUtils,HttpResponse  ⍝ create the response container

      :Case 'HTTPHeader'
        resp.CongaHttpHeader dat ⍝ process the message headers

      :Case 'HTTPBody'
        resp.CongaHttpBody dat  ⍝ process the message body

      :Case 'HTTPChunk'
        resp.CongaHttpChunk dat  ⍝ process the a chunk of the message

      :Case 'HTTPTrailer'
        resp.CongaHttpTrailer dat  ⍝ process the message trailers

      :EndSelect
    :EndIf
  :Until done∨resp.IsComplete ⍝ do we have the complete resposnse?
:EndIf
```

At this point we have all of the response data in public fields of the `HttpResponse` instance.

```
      resp.(HttpStatus HttpStatusText)
 200  ok

      resp.Headers   ⍝ HTTP headers
 cache-control     private
 content-type      text/html
 content-encoding  gzip
 vary              Accept-Encoding
 server            Microsoft-IIS/8.5
 set-cookie        ASPSESSIONIDCATBDSCB=ANDNFICBLDIEHFOOINJHFOAL; path=/
 x-powered-by      ASP.NET
 date              Tue, 27 Jun 2017 12:34:02 GMT
 content-length    370
```

```
      resp.Data  A response data (edited for presentation here)
<html>
<head>
<meta http-equiv="Content-Language" content="en-us">
<meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
<title>Some really cool place</title>
</head>
<body bgcolor="#FFFFFF">
<p align="center"><img src="images/BDS.jpg"></p>
<p align="center"> </p>
<p align="center"><b><font face="Arial">Some really cool place...coming soon!
</font></b></p>
</body>
</html>
```