# z/OS V2R3 Client Web Enablement Toolkit: More Tools to Transform Your z/OS Applications into RESTful Clients the Easy Way

Steve Warren, IBM
z/OS Client Web Enablement Toolkit Tech Lead
Email: swarren@us.ibm.com
:@StevieWarr2

August 9, 2017

## Agenda

- Overview of Toolkit
- Basic usage of HTTP Enabler
- New toolkit features
  - REXX support
  - HTTP streaming send and receive (V2R3)
  - AT-TLS interoperability support
  - Enhanced HTTPS functionality
  - Support for direct HTTP tracing to data sets or zFS files
  - JSON parser performance improvements for very large JSON text
  - Support for the HEAD HTTP request method

## Overview of Toolkit

## z/OS serving REST APIs

- z/OS platform has for years been labeled “the server of servers” and houses much of the world’s most critical data.
- Enhancements to the z/OS Web serving space through the years have allowed this mammoth workhorse and repository of data to be more easily accessible to other systems.

## What about z/OS as a REST client?

# Introducing the z/OS Client Web Enablement Toolkit!

**The z/OS client web enablement toolkit provides a set of lightweight application programming interfaces (APIs) to enable traditional, native z/OS programs to participate in modern web services applications.**

- Pieces of the toolkit:
  - A z/OS HTTP/HTTPS protocol enabler to externalize HTTP and HTTPS client functions in an easy-to-use generic fashion for user's in almost any z/OS environment
  - A z/OS JSON parser which parses JSON coming from any source, builds new JSON text, or adds to existing JSON text.
- The toolkit allows its two parts to be used independently or combined together.
  - Payload processing is separate from communication processing.
- The interfaces are intuitive for people familiar with other HTTP enabling APIs or other parsers
- Easy for newbies

## General programming toolkit environment

- Runs in just about any address space
  - Code runs in user’s address space
- Supports both authorized and un-authorized callers
- Easy API suite provided
- Multi-language support
  - Include files supplied for C, COBOL, PL/I, Assembler, REXX
  - Multi-language samples provided

## z/OS HTTP/HTTPS Protocol Enabler Details

## Usage & Invocation – z/OS HTTP Services

- Provides similar functionality to existing open-source libcurl HTTP/HTTPS interface
  - Interface is very similar
  - Underlying code is z/OS-specific and not ported in any way

## HTTP features supported by toolkit

- HTTPS connections
- HTTP cookies management
- Proxies
- URI redirection
- Basic client authentication
- Chunked encoding

## HTTP Services execution environment

- Specific requirements for the HTTP portion of the toolkit:
  - Supports task mode, non-cross-memory callers
  - Execution key 1 thru 15 allowed
  - OMVS segment required for address space using HTTP enabler
- Recovery recommended by caller

## Demo 1

## Demo 2

## Structure of an HTTP Toolkit Application

## Connections / Requests

- The HTTP/HTTPS enabler portion of the toolkit encompasses two major aspects of a web services application:
  - The ***connection*** to a server
  - The ***request*** made to that server along with the response it returns

## HTTP Connections

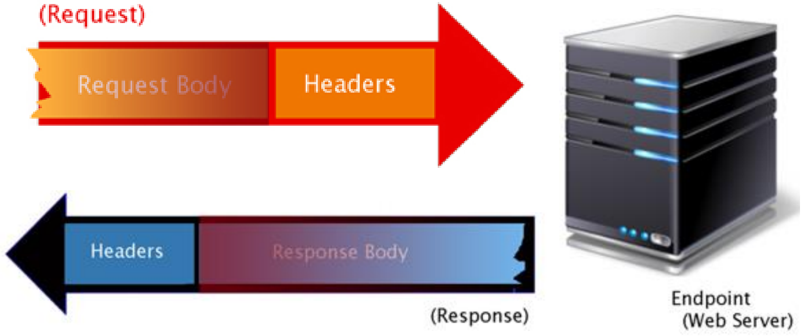
- A connection is simply a socket (pipeline) between the application and the server.
- Must be established first before a request can flow to the server.
- Many options available for connection including:
  - SSL/TLS
  - Local IP address specification
  - IP Stack
  - Timeout values

## Steps to create an HTTP Connection

- Initialize a connection (HWTHINIT)
  - Obtain workarea storage for the connection
- Set one or more connection options (HWTHSET)
  - One option at a time
- Make the actual connection (HWTHCONN)
  - Creates the socket to the specified server

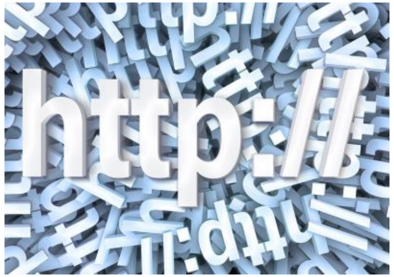
# HTTP Request Overview

- A client makes an HTTP request to a server (endpoint)
  - This HTTP request will typically be one of these types:
    - GET (read existing resource)
    - PUT (write/update existing resource)
    - POST (write new resource)
    - DELETE (remove existing resource)
  - May send Request Headers
  - May send Request Body (PUT and POST)



- The endpoint returns an HTTP response
  - Response consists of status (1xx, 2xx, 3xx, 4xx, 5xx)
  - Response headers
  - Response body (most requests)

## HTTP Requests



- An HTTP request sent over an existing connection
  - Targets a particular resource at the domain established by the connection
  - An HTTP GET, PUT, POST or DELETE is specified as the request method
- Requests not tightly-coupled to a connection. The same request can be sent over different connections
- Response callback routines (exits) can be set prior to the request to handle returned response headers and response body.

## Steps to create an HTTP Request

- Initialize a request (HWTHINIT)
  - Obtain workarea storage for the request
- Set one or more request options (HWTHSET)
  - One option at a time
- Send the request over a specified connection (HWTHRQST)
  - Flows the HTTP REST API call over the connection (socket) and then receives the response

## HTTP Connection Details

## Where do I want to connect?

- REST API specification details the location where the request must target
- Here are some examples:
  - Google® Maps Directions API specifies:
    - https://maps.googleapis.com/maps/api/directions/*outputFormat?paramaters*
  - Yelp® Search API specifies:
    - http://api.yelp.com/v2/search?*searchParms*
  - FAA Airport Service API specifies:
    - http://services.faa.gov/airport/status/*airportCode*

# Format of a URI (URL) in the world of HTTP

A Uniform Resource Identifier (URI) is the way a REST API is represented. A simplified syntax of a URI is:

```
    scheme://host[:port]          [/]path[?query][#fragment]
    |--------------------------------|    |---------------------------------------|
      where and how to connect?    what is the particular request?
      Connection portion of URI           Request portion of URI
```

**scheme** – HTTP (unencrypted) or HTTPS (encrypted) (optional)
host – hostname or IPv4 or IPv6 address (e.g. www.ibm.com)
port – an optional destination port number (80 is default for HTTP scheme, 443 for HTTPS scheme)
path – an optional hierarchical form of segments (like a file directory) which represents the resource to perform the HTTP request method against
query – an optional free-form query string to allow for the passing of parameters
fragment – an optional identifier providing direction to a secondary resource

## Setting Location of Connection

- HWTH_OPT_URI (required) – valid values are either a v4 or v6 IP address, or hostname
    - Examples:
        - http://192.168.0.1
        - http://[2001:1890:1112:1::20]
        - http://www.example.com
    - HTTP Scheme is optional
- HWTH_OPT_PORT (optional) – value specifying which remote port number to connect to, instead of the one specified in the URL or the default HTTP or HTTPS port.
    - Default for HTTP is 80
    - Default for HTTPS is 443

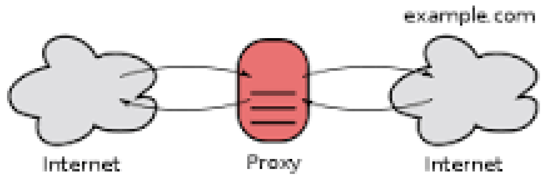
## Setting Source Location of Connection

- HWTH_OPT_IPSTACK – Optional value 1 to 8 character z/OS TCP/IP stack to be used by the connection
  - Procname of TCP/IP (useful when installation has more than one TCP/IP stack running)
- HWTH_OPT_LOCALIPADDR – Optional outgoing local IP address
- HWTH_OPT_LOCALPORT – Optional outgoing local port

- Definitions also useful especially when security definitions are in place for specifically defined addresses and ports
  - Application limited by existing security profiles and definitions that are already in effect on the system where the application resides.
    - z/OS Communications Server NetAccess, in conjunction with security profiles defined using the SERVAUTH class, can be used to control network access authority, TCP/IP stack access authority, port access authority, and more.

## HTTP Services – Proxy Options

- HWTH_OPT_PROXY – set the HTTP proxy to user. Specified the exact same as HWTH_OPT_URI above.
- HWTH_OPT_PROXYPORT – specify the proxy port to connect to. Specified the exact same as HWTH_OPT_PORT above



## Setting other connection options

- HWTH_OPT_SNDTIMEOUTVAL – Sending timeout value
- HWTH_OPT_RCVTIMEOUTVAL – Receiving timeout value
- HWTH_OPT_HTTP_VERSION – which version of HTTP do you want to use over this connection
  - HTTP_VERSION_NONE
  - HTTP_VERSION_1_0
  - HTTP_VERSION_1_1

# HTTP Services – SSL Options

- SSL support options include:
  - HWTH_OPT_USE_SSL – tells toolkit to attempt SSL negotiation explicitly
  - HWTH_OPT_SSLVERSION – sets the SSL versions to be supported by this HTTP request. More than one version may be selected. (e.g. TLS1.2, TLS1.1, TLS1.0, SSLv3)
  - HWTH_OPT_SSLKEYTYPE – Specifies the manner the key will be supplied to this HTTPS request. The following constants are provided:

    SSLKEYTYPE_KEYDBFILE – key store is specified key database file

    SSLKEYTYPE_KEYRINGNAME – key store is a security product managed keyring.
  - HWTH_OPT_SSLKEY – Specifies the value of the key. The value specified depends on the value set by SSLKEYTYPE.

    For SSLKEYTYPE_KEYDBFILE - represents path and name of the key database file name

    For SSLKEYTYPE_KEYRINGNAME – represents the SAF key ring name or PKCS#11 token
  - HWTH_OPT_SSLKEYSTASHFILE – specifies the stash file of the key database file. Only valid if SSLKEYTYPE_KEYDBFILE is specified. Ignored in all other cases.
  - HWTH_OPT_SSLCLIENTAUTHLABEL – optional label that represents a client certificate if SSL client authentication is requested by the server.

## HTTP Services – Redirect Options

- Redirect options include:
  - HWTH_OPT_MAX_REDIRECTS – maximum number of redirects to follow for a given request.
  - HWTH_OPT_XDOMAIN_REDIRECTS – are cross-domain redirects allowed?
  - HWT_OPT_REDIRECT_PROTOCOLS – do you allow the HTTP/HTTPS protocol to be upgraded and/or downgraded on a redirect?

## HTTP Services – Cookie Options

- Cookie support options include:
  - HWTH_OPT_COOKIETYPE – sets cookie handling type

    COOKIETYPE_NONE – cookie engine not activated

    COOKIETYPE_SESSION – cookie engine enabled – cookies automatically sent, but end when connection ends

    COOKIETYPE_PERSIST - cookie engine enabled – cookies automatically sent, cookies saved to output buffer when connection endsin

  - HWTH_OPT_COOKIE_INPUT_BUFFER – specifies input cookie data store
  - HWTH_OPT_COOKIE_OUTPUT_BUFFER – specifies output cookie location for a cookietype of COOKIETYPE_PERSIST

## Example of establishing a connection

- Initialize a connection (HWTHINIT)
- Set one or more connection options (HWTHSET)
- Connect to the web server (HWTHCONN)

## HTTP Request Details

## What resource and parameters do I want to invoke?

- REST API specification details the location where the request must target
- Here are some examples:
  - Google® Maps Directions API specifies:
    - https://maps.googleapis.com/maps/api/directions/*outputFormat?paramaters*
  - Yelp® Search API specifies:
    - http://api.yelp.com/v2/search?*searchParms*
  - FAA Airport Service API specifies:
    - http://services.faa.gov/airport/status/*airportCode*

# Format of a URI (URL) in the world of HTTP

A Uniform Resource Identifier (URI) is the way a REST API is represented. A simplified syntax of a URI is:

```
scheme://host[:port]            [/]path[?query][#fragment]
|--------------------------------|  |----------------------------------------|
  where and how to connect?        what is the particular request?
  Connection portion of URI             Request portion of URI
```

**scheme** – HTTP (unencrypted) or HTTPS (encrypted) (optional)

host – hostname or IPv4 or IPv6 address (e.g. www.ibm.com)

port – an optional destination port number (80 is default for HTTP scheme, 443 for HTTPS scheme)

path – an optional hierarchical form of segments (like a file directory) which represents the resource to perform the HTTP request method against

query – an optional free-form query string to allow for the passing of parameters

fragment – an optional identifier providing direction to a secondary resource

## Setting request path, input parameters, method

- HWTH_OPT_URI (required) – The name or resource (URN path portion) of the URI. The query and fragment portions of a URI may also be present.
  - Examples:
    - /systems/z/
    - /over/here?name=abc#frag1
- HWTH_OPT_REQUEST (required) – which HTTP CRUD request method does the request want to use
  - HWTH_HTTP_REQUEST_GET
  - HWTH_HTTP_REQUEST_PUT
  - HWTH_HTTP_REQUEST_POST
  - HWTH_HTTP_REQUEST_DELETE

## HTTP Services – Chunked encoding

- **Responses with chunked encoding present**
  - Toolkit supports the chunked encoding data transfer method (Transfer-encoding: chunked).
  - Automatically de-chunks data sent from the server using the chunked encoding method.
    - The response body exit does not need to handle the various chunks; rather, the data is delivered to the exit already decoded.
    - If the chunked data contains trailer headers, the header exit will be invoked (once for each trailer header) prior to this routine receiving control.
  - Note: The toolkit ignores chunk extensions

## Other HTTP request options

- HTTP authorization options:
  - HWTH_OPT_HTTPAUTH – Do I want HTTP basic client authentication?
  - HWTH_OPT_USERNAME and HWTH_OPT_PASSWORD must be set if basic client authentication is selected.
- HTTP request body and response body translate functions
  - HWTH_OPT_TRANSLATE_REQBODY – translate request body from EBCDIC to ASCII automatically.
  - HWTH_OPT_TRANSLATE_RESPBODY – translate response body from ASCII to EBCDIC automatically.

## Example of establishing a request

- Initialize a request (HWTHINIT)
- Set one or more request options (HWTHSET)
- Send the request to the web server (HWTHRQST)

## Example of initializing a request

```
HandleType = HWTH_HANDLETYPE_REQUEST
address hwthttp "hwthinit ",
                "ReturnCode ",
                "HandleType ",
                “ReqHandle ",
                "DiagArea."
```

- A request instance has been created.
- As many request instances as you would like can be created
- A request is married with a connection at the HWTHRQST API call

# Example of setting some request options

```
/***************************************************************/
/* Set HTTP Request method.                                    */
/* A GET request method is used to get data from the server.   */
/***************************************************************/
address hwthttp "hwthset ",
                "ReturnCode ",
                "ReqHandle ",
                "HWTH_OPT_REQUESTMETHOD ",
                "HWTH_HTTP_REQUEST_GET ",
                "DiagArea."
*****************************************************************/
* Set the request URI                                           */
*  Set the URN URI that identifies a resource by name that is   */
*    the target of our request.                                 */
*****************************************************************/
requestPath = '/airport/status/'||airportCode
address hwthttp "hwthset ",
                "ReturnCode ",
                "ReqHandle ",
                "HWTH_OPT_URI ",
                "requestPath ",
                "DiagArea."
```

- All the request options can be set before issuing the HWTHRQST service
- Once the request is completed, the request options can be altered if desired before next send request service.

## Example of setting a request header

```
acceptJsonHeader = 'Accept:application/json'
/************************************************************************
/* Create a brand new SList and specify the first header to be an      *
/* "Accept" header that requests that the server return any response  *
/* body text in JSON format.                                           *
/************************************************************************
address hwthttp "hwthslst ",
                "ReturnCode ",
                "ReqHandle ",
                "HWTH_SLST_NEW ",
                "SList ",
                "acceptJsonHeader ",
                "DiagArea.“
/*********************************************************/
/* Set the request headers with the just-produced list */
/*********************************************************/
address hwthttp "hwthset ",
                "ReturnCode ",
                "ReqHandle ",
                "HWTH_OPT_HTTPHEADERS ",
                "SList ",
                "DiagArea."
```

- After the first HWTH_SLST_NEW function, specify HWTH_SLIST_APPEND to add more headers to the SLST if desired.

## Example of setting more request options

```
/*****************************************************************/
/* Tell the toolkit to translate the body from ASCII to EBCDIC   */
/*****************************************************************/
address hwthttp "hwthset ",
                "ReturnCode ",
                "ReqHandle ",
                "HWTH_OPT_TRANSLATE_RESPBODY ",
                "HWTH_XLATE_RESPBODY_A2E ",
                "DiagArea."
/*****************************************************/
/* Set the variable for receiving response body   */
/*****************************************************/
say 'Set HWTH_OPT_RESPONSEBODY_USERDATA for request'
address hwthttp "hwthset ",
                "ReturnCode ",
                "ReqHandle ",
                "HWTH_OPT_RESPONSEBODY_USERDATA ",
                "ResponseBody ",
                "DiagArea."
```

- When the HWTHRQST service completes, the response body will be in EBCDIC.

## Example of finally issuing the REST API call

```
/***********************************/
/* Call the HWTHRQST toolkit api.  */
/***********************************/
address hwthttp "hwthrqst ",
                "ReturnCode ",
                "ConnHandle ",
                "ReqHandle ",
                "HttpStatusCode ",
                "HttpReasonCode ",
                "DiagArea."
```

- When this API call completes, the request and response have also completed. The final status of the call is returned in the HttpStatusCode (eg. 200) and HttpReasonCode (eg. “OK”)
  - Non-REXX
    - Header and Response body exits have already been driven
  - REXX
    - Response Header and Response body variables can be consulted
- A reset or terminate of the request will wipe out any set values

## z/OS Client Toolkit HTTP Language Support

Include files and sample programs provided in:

- C
- COBOL
- PL/I
- Assembler (Include file only)
- REXX
  - sample delivered via APAR OA50659 on V2R1 and V2R2

# New REXX support details

## New REXX Host Commands for Toolkit

- Two new host command environments available in REXX
  - HWTJSON
  - HWTHTTP
- Runs in the following REXX environments:
  - TSO
  - ISPF
  - System REXX
  - z/OS UNIX
  - ISV-provided REXX environments
- How to get the host command environment
  - V2R3 and higher (non-ISV-provided environments)
    - host command environments built-in
  - V2R2 and lower (and ISV-provided environments on any release)
    - Host command environment dynamically added by adding hwtcalls(‘on’) to the REXX exec

## Toolkit REXX Support general syntax

- HTTP enabler syntax

```
address hwthttp 'service_name returncode service_args... diagnostic_stem.'
```

- z/OS JSON parser syntax

```
address hwtjson 'service_name returncode service_args... diagnostic_stem.'
```

- Host command return code (special variable rc in REXX) should be 0
  - >0 toolkit service not called, verify command arguments
  - <0 probably an abend code, service may or may not have been called
- The HTTP enabler require z/OS UNIX signals.
  - May wish to issue `call syscalls ‘SIGOFF’` depending on your environment

## New REXX-only toolkit services

- HWTCONST (get toolkit constants)
  - Initializes pre-defined variables in the current REXX variable pool
  - Useful for using the symbolic names as defined in the documentation (such as checking for return codes, setting a particular option, etc..)
  - `address hwthttp ‘hwtconst returnCode diag.’`
  - The variable HWT_CONSTANTS is set to a string containing the names of all the variables set
    - Useful for procedures to expose names (`procedure x expose(hwt_constants)` )
- HWTJESCT (encode/decode escape sequences in JSON text)
  - Transforms non-conforming JSON text (not properly escaped) into conforming JSON text (encode). Decode does the reverse.
  - `address hwtjson ‘hwtjesct returnCode func srcTxt trgTxt diag‵`
    - Where func is either `HWTJ_ENCODE` or `HWTJ_DECODE`

## Toolkit service not available in REXX

- HWTJGNUV (get number value)
  - REXX has no concept of integers or floating point numbers

## Example how to init HTTP connection in REXX

```
HandleType = HWTH_HANDLETYPE_CONNECTION
address hwthttp "hwthinit ",
                "ReturnCode ",
                "HandleType ",
                "ConnectHandle ",
                "DiagArea."
```

- A connection instance has been created.
- More than one connection can be initialized per address space (if the user has set his dubbing defaults to dub process)
  - But only one connection can be active at a time in an MVS task  (Setup MVS Signals only allows one signal setup to be allowed per process)

## Example how to set connection options in REXX

```
/*********************************************************************/
/* Set URI for connection to the Federal Aviation Administration (FAA) */
/*********************************************************************/
ConnectionUri = 'http://services.faa.gov'
ReturnCode = -1
DiagArea. = ''
address hwthttp "hwthset ",
                "ReturnCode ",
                "ConnectHandle ",
                "HWTH_OPT_URI ",
                "ConnectionUri ",
                "DiagArea."
/*********************************************************************** */
/* Set HWTH_OPT_COOKIETYPE                                              */
/*   Enable the cookie engine for this connection.                      */
/*   Any "eligible" stored cookies will be resent to the host on subsequent */
/* interactions automatically.                                          */
/************************************************************************/
ReturnCode = -1
DiagArea. = ''
address hwthttp "hwthset ",
                "ReturnCode ",
                "ConnectHandle ",
                "HWTH_OPT_COOKIETYPE ",
                "HWTH_COOKIETYPE_SESSION ",
                "DiagArea."
```

## Example showing connecting to HTTP server in REXX

```
/*********************************/
/* Call the HWTHCONN toolkit api  */
/*********************************/
ReturnCode = -1
DiagArea. = ''
address hwthttp "hwthconn ",
                "ReturnCode ",
                "ConnectHandle ",
                "DiagArea."
```

- The connection stays persistent from a toolkit perspective
  - Timed-out connections (sockets) will automatically be reconnected at the time of a request if necessary
- A disconnect, reset or terminate of the connection will disconnect the established connection

# Processing HTTP Response Headers in REXX

- All HTTP response headers are optionally stored in a REXX stem variable
  - Set the HWTH_OPT_RESPONSEHDR_USERDATA option to the name of the stem variable.
- Upon return from the HWTHRQST (send request service), the following information is returned regarding the response headers received:
  - the number of response headers received is stored in the .0 stem
  - the name of each response header is stored in the stemname.x stem variable, where x is the from 1 to the number of response headers received.
  - the value for each response header is stored in the stemname.x.1 stem variable, where is x is from 1 to the number of response headers received.

## Sending a Request Body in REXX

- The main data sent on a PUT or POST method to an HTTP REST server is usually sent in the request body
- Set the HWTH_OPT_REQUESTBODY option with the simple name of the variable which contains the request body.

## Receiving the Response Body in REXX

- The main data returned from an HTTP REST server is usually sent as the response body
- Set the HWTH_OPT_RESPONSEBODY_USERDATA option with the simple name of the variable which will hold the response body data.
- REXX limitation of 16Mb applies for max response body to be received in REXX

## New HTTP Streaming Support

## Why HTTP Streaming?

- **A matter of size**
  - HTTP protocol has no limit on body size
  - Either body may be “very” large
  - What if a body is so large that it is not feasible to hold it in memory at a given point in time?
- **Awareness of body size**
  - HTTP protocol allows for cases when body size is known, allowing its transfer “all at once”
  - What if the body size is not known up front?
    - HTTP protocol allows for a body to be transferred as an ordered sequence of pieces (“gradually”), allowing for the end of the body to be discovered/conveyed.
- **Transparency**
  - Does a user want to understand and handle metadata artifacts which might be required by a gradual body transfer?
  - Can the toolkit (user agent) efficiencies be improved?

## HTTP Enabler capabilities before/after streaming

- **Before**
  - "all at once" Request/Response Body (moderate size)
  - "gradual" Response Body (moderate size)
    - Toolkit is inefficient in its processing of gradual (chunk-encoded) Response Bodies
      - De-chunks in place using a "repeated-shift-left" approach
      - Not suitable for larger body sizes
- **After**
  - "gradual" Request Body (any size)
  - "all at once" Response Body (of arbitrarily large size)
  - "gradual" Response Body (of arbitrarily large size)

## Sending HTTP Streaming Use Case

- Need to send an arbitrarily large piece of data to the server (as a request body)
- Cannot hold all of the data at once in storage
- Want to spoon-feed moderate-sized pieces of the data to the toolkit
  - Toolkit can chunk-encode it on my behalf (if necessary) as part of the transfer

User needs to:

- Give the data pieces to the toolkit in order
  - Wants flexibility to provide them at varying addresses with varying sizes (since that is how they may be made available to the user)
- Indicate "that's all of my data" to the toolkit
- Tell the toolkit to cancel the remainder of any in-progress send
- Be notified by the toolkit if anything goes wrong along the way

# Receiving HTTP Streaming Use Case

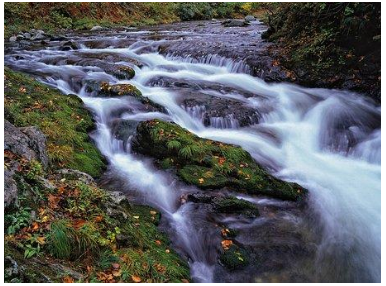
- Need to receive an arbitrarily large piece of data from the server (as a response body)
- May not be able to hold all of the data at once in storage
  - Size may be unpredictable
- Want the toolkit to spoon-feed the application moderate-sized pieces of the data
  - Data should be devoid of any chunk-encode metadata which the toolkit may have encountered during transfer

User expects to:

- Receive the pieces in order and efficiently written directly into the user's buffers
  - Wants flexibility to provide the buffers at varying addresses with varying sizes
- Be indicated by the toolkit "that's all of my data"
- Tell the toolkit to cancel the remainder of any in-progress receive
- Be notified by the toolkit if anything goes wrong along the way

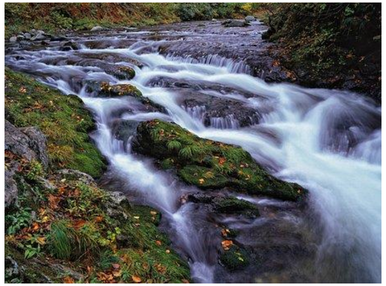
## Usage of HTTP Streaming (Send to server)

- New Streaming Send Exit option settable by HWTHSET
  - If valid exit address is set for `HWTH_OPT_STREAM_SEND_EXIT`, streaming send will be in effect
- New Streaming user data option settable by HWTHSET
  - Set `HWTH_OPT_REQUESTBODY_USERDATA` to provide exit with an optional buffer of user data to be passed to the streamed send callback routine.
- The toolkit will drive the streaming send exit in a loop to process the data
  - Send exit will specify:
    - address and length of the data buffers to be sent
    - a status value to communicate the state of the data being sent



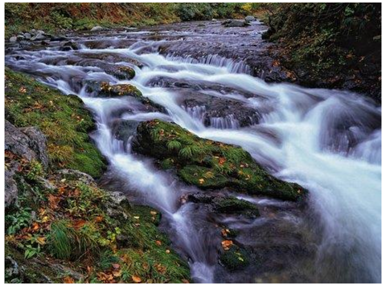
## Usage of HTTP Streaming (Receive from server)

- New settable options for Streaming Receive exit and Streaming Receive exit user data
  - Note that the `HWTH_OPT_STREAM_RECEIVE_EXIT` is mutually exclusive with the existing `HWTH_OPT_RESPONSEBODY_EXIT` option.
- The toolkit will drive the streaming receive exit in a loop to process the data
  - Receive exit will specify:
    - address and length of the data buffers to be populated by the data received
    - a status value to communicate the state of the data being received
  - Receive exit is notified when all the data has been received



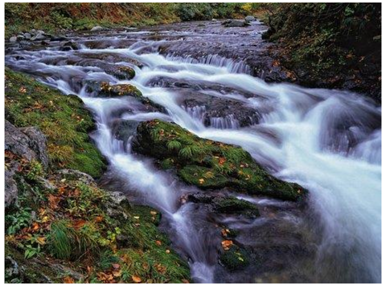
## Miscellaneous Streaming Notes

- No REXX support for streaming
- Data moves minimized as much as possible
  - In some cases, the data is written directly from the socket to the user buffer.
- Vector list required by Streaming Send exit compatible with the readv() z/OS XL C/C++ library function (Read data from a file and store into a set of buffers)
- Vector list returned in Streaming Receive exit compatible with the writev() z/OS XL C/C++ library function (Write data from a set of buffers into a file)



## Streaming samples

- New streaming samples now available
  - C and COBOL examples provided
  - Illustrate sending a file to a well-known HTTP server and receiving it back
  - Available via APAR OA53922 on V2R3



## AT-TLS / Toolkit Interoperability Support

## HTTP Services – AT-TLS / Toolkit Interoperability

- Application Transparent – TLS is basically stack-based TLS
  - TLS process performed in TCP layer (via System SSL) without requiring any application change (transparent)
  - AT-TLS policy specifies which TCP traffic is to be TLS protected based on a variety of criteria
    - Local address, port
    - Remote address, port
    - z/OS userid, jobname
    - Time, day, week, month
  - Gives network administrators greater control over the security requirements of network applications rather than individual applications

AT-TLS

# HTTP Services – AT-TLS / Toolkit Interoperability

- Toolkit is now AT-TLS aware (with APAR OA50957 installed)
  - Application *does not specify* SSL/TLS options directly within toolkit application?
    - **AT-TLS policy upgrades connection to SSL/TLS?**
      - Toolkit will treat the requests over this connection as HTTPS requests
      - All cookies and redirect processing will be now operate as an HTTPS request.
    - **AT-TLS policy does not upgrade connection or no policy in effect?**
      - Business as usual. Request will operate as HTTP
  - Application *specifies* SSL/TLS directly within toolkit application?
    - **AT-TLS policy upgrades connection to SSL/TLS?**
      - Toolkit rejects the request. Network configuration and application are in conflict.
    - **AT-TLS policy does not upgrade connection or no policy in effect?**
      - Business as usual. SSL/TLS credentials will be specified by the application. If successful handshake, then request will operation as HTTPS.

**AT-TLS**

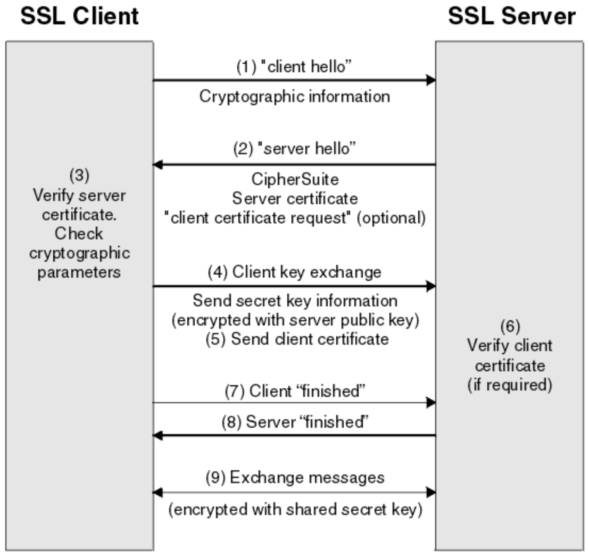
## Enhanced HTTPS Functionality

## New SSL Cipher Specs support

- When a secure connection is established, the client and server negotiate the cipher to use for the connection (RFC5246). These ciphers help determine how the data will be encrypted and decrypted.
- The web server has an ordered list of ciphers, and the first cipher in the list that is supported by the client is selected.
- New `HWTH_OPT_SSLCIPHERSPECS` option allows the application to specify a list of 4-character cipher definitions
  - Should be ordered by preference of use
  - Requires `HWTH_OPT_USE_SSL` option to be set to `HWTH_SSL_USE` (application-initiated SSL connection)
- Allows the client application to replace the default list of acceptable cipher specifications with its own list
- Available in APAR OA53546 on V2R1 and higher

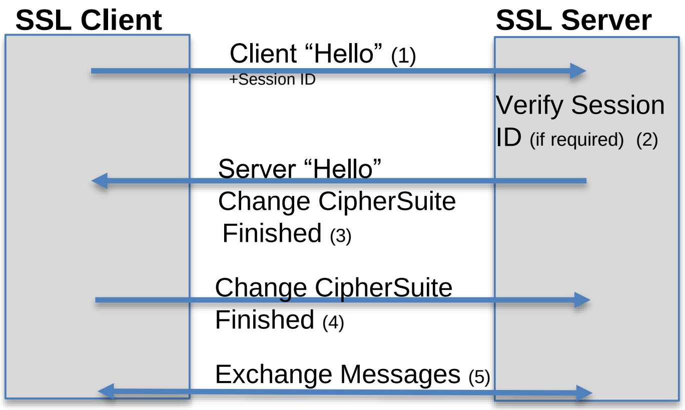
# New SSL TLS 1.2 optimization support

- Full TLS/SSL handshake (prior to TLS 1.2)
  - High latency
  - Two round-trips required
  - Expensive computation to exchange keys or sign and verify certificates



# New SSL TLS 1.2 optimization support

- Abbreviated (short) handshake (TLS 1.2)
  - The full handshake is required at least once.
  - The full handshake results in server sending a **session ID** back to the client.
  - This ID is cached on the server and by the toolkit (client-side)
  - If new request is made to the same server and the connection is no longer there, the toolkit attempts to send this cached session ID as part of the new handshake.
  - If the server accepts this session ID, the server quickly completes the handshake, bypassing most of the full handshake steps.
  - If the server does not accept the session ID, a full handshake will result.



## New SSL TLS 1.2 optimization support …

- Available on z/OS V2R2 and higher
  - Requires APAR OA53546 to be installed for application-initiated SSL connections
    - The toolkit will use the abbreviated handshake whenever it is possible to resume a previously established secure connection.
- Connections using AT-TLS can also avail themselves of this optimization automatically when running on V2R2 or higher
- Toolkit optimization will only be available if a connection has not been disconnected. Use cases include:
  - A server times out a connection. A request is then attempted over this timed-out connection.
  - A server sends a Connection Closed response header (or fails to specify Keep-Alive (HTTP 1.0)).  A request is then attempted over this closed connection.

## HTTP Tracing Output to a DD

## New HTTP tracing to DD support

- New `HWTH_OPT_VERBOSE_OUTPUT` option allows specification of a DD where HTTP trace output is to be directed
- Used in conjunction with the existing `HWTH_OPT_VERBOSE` option
- The DD name above must represent either:
  - a pre-allocated traditional z/OS data set which is a physical sequential (DSORG=PS) with a record format of unblocked variable (RECFM=V) or Undefined (RECFM=U) and expandable (non-zero primary and secondary extents). The DD must also specify a DISP=OLD disposition.
  - a zFS or HFS file.

```
t: An error occurred: Certificate validation error
t: Reason code: 8
t: Return code: -1
t: Service: 22
t: Service Instance: 0
```

## New HTTP tracing to DD support

```
address hwthttp "hwthset ",
                "ReturnCode ",
                "ConnHandle ",
                "HWTH_OPT_VERBOSE ",
                "HWTH_VERBOSE_ON ",
                "DiagArea."

traceDD = ‘MYTRACE’
/* Allocate the data set here */
address hwthttp "hwthset ",
                "ReturnCode ",
                "ConnectionHandle ",
                "HWTH_OPT_VERBOSE_OUTPUT ",
                "traceDD ",
                "DiagArea."
```

- When a connect, sendRequest or disconnect are issued, detailed trace will be written to the data set specified by the MYTRACE DD.

## JSON parsing performance improvements (large streams)

## Parsing performance improvements for very large JSON

- No external changes.
- Very large JSON text streams parse much faster (HWTJPARS) than before
  - z/OS JSON parser much smarter about the management of the internal JSON mapping of the JSON text stream.
  - For very large JSON bodies, the internal representation can now span multiple obtained storage areas.
- Creating new JSON entries (HWTJCREN) occur significantly faster
  - The insertion point for adding an new entry to an object or array is learned significantly faster than before, making insertion orders of magnitude faster than before when there are many entries already in the object or array.

{"MyJSON":
{"xyz":123}
}

## Support for HEAD HTTP Request Method

## Support for HEAD request method

- The HEAD method is identical to GET except that the server does not return a message-body in the response. The metadata contained in the HTTP headers in response to a HEAD request will be identical to the information sent in response to a GET
- HWTHSET can now be invoked using a new constant value `HWTH_HTTP_REQUEST_HEAD` for setting the `HWTH_OPT_REQUESTMETHOD` option.

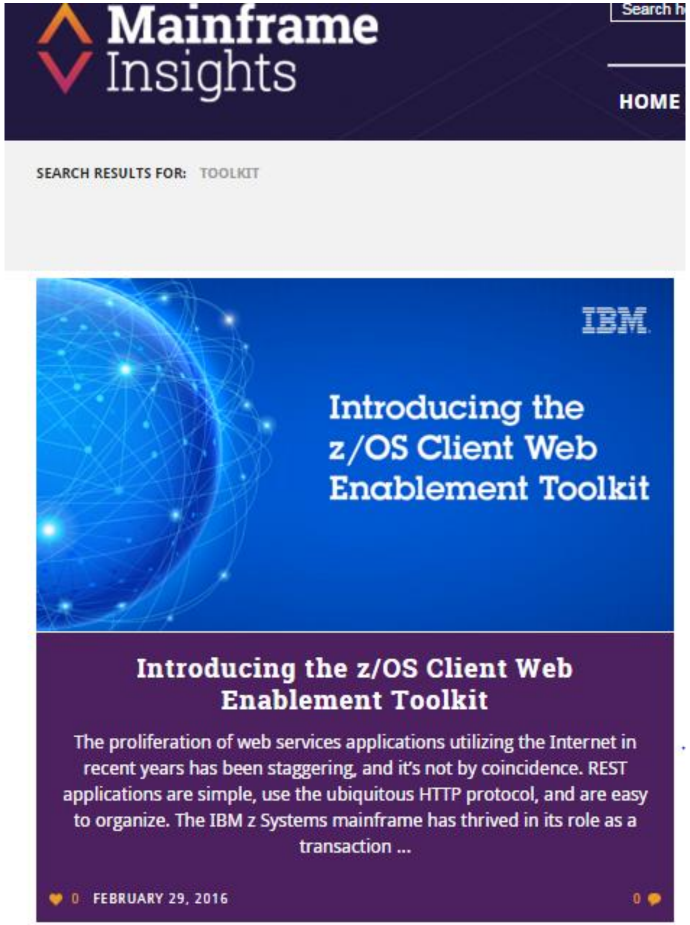
## Where to go for more z/OS Client Web Enablement Toolkit information

## Come to the SHARE Lab

- **Writing a z/OS application leveraging REST APIs**
  - Friday, August 11th, 8:30am, Room 557
- Learn how to write a couple of simple applications which leverage REST APIs
  - Learn the basic functions of the toolkit
  - Learn how to debug a few common problems encountered when invoking a REST API

## Toolkit Reference Materials

- **z/OS 2.2 MVS Programming: Callable Services for High-Level Languages**
  - Complete toolkit documentation
- **z/OS 2.2 MVS System Messages, Volume 6 (GOS – IEA)**
  - Toolkit message documentation
- **z/OS 2.2 MVS System Codes**
  - Toolkit abend ’04D’x documentation



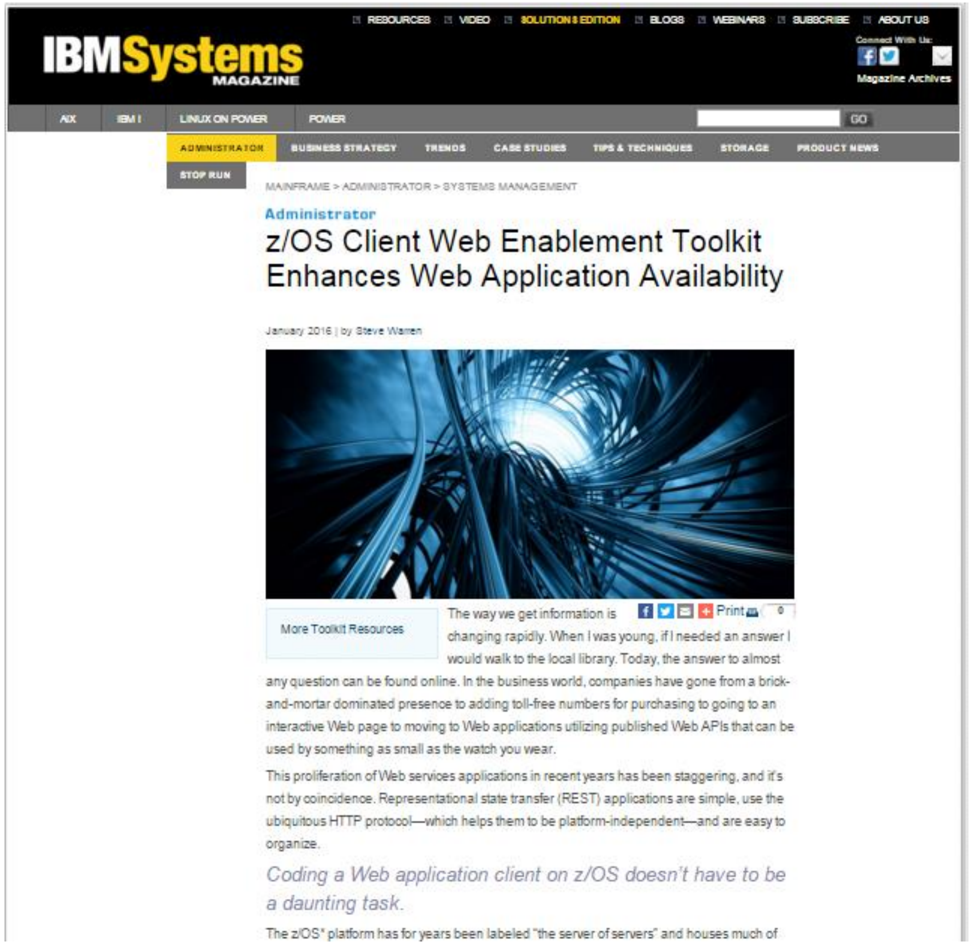
SEARCH RESULTS FOR: TOOLKIT

Introducing the z/OS Client Web Enablement Toolkit

The proliferation of web services applications utilizing the Internet in recent years has been staggering, and it's not by coincidence. REST applications are simple, use the ubiquitous HTTP protocol, and are easy to organize. The IBM z Systems mainframe has thrived in its role as a transaction ...

0 FEBRUARY 29, 2016 0

- **mainframeinsights.com**
  - Search for toolkit



Administrator

z/OS Client Web Enablement Toolkit Enhances Web Application Availability

January 2016 | by Steve Warren

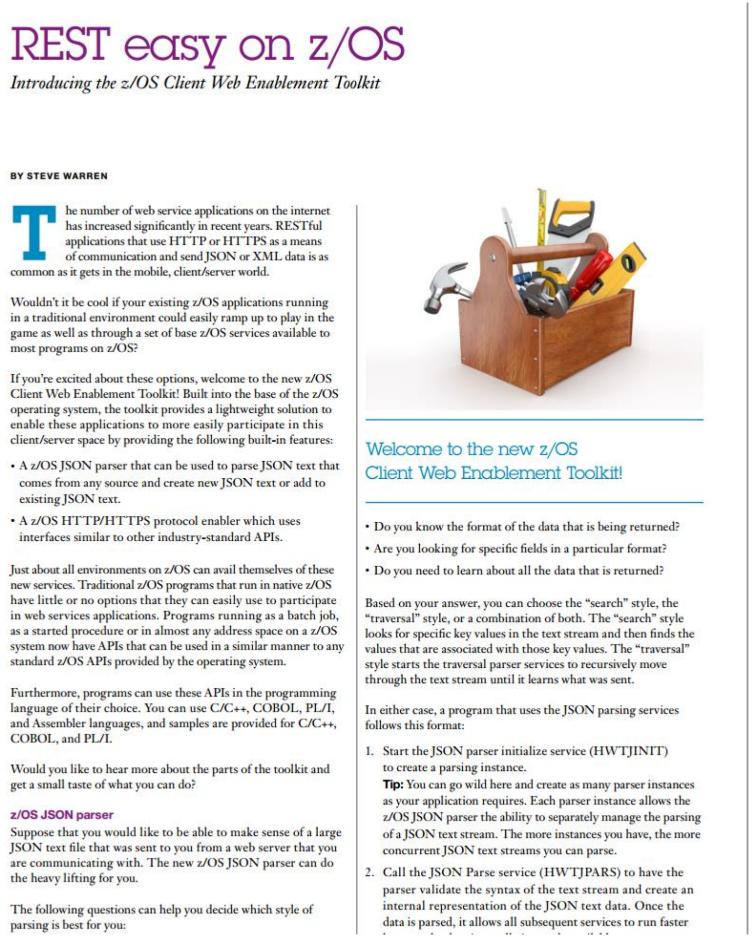
The way we get information is changing rapidly. When I was young, if I needed an answer I would walk to the local library. Today, the answer to almost any question can be found online. In the business world, companies have gone from a brick-and-mortar dominated presence to adding toll-free numbers for purchasing to going to an interactive Web page to moving to Web applications utilizing published Web APIs that can be used by something as small as the watch you wear.

This proliferation of Web services applications in recent years has been staggering, and it's not by coincidence. Representational state transfer (REST) applications are simple, use the ubiquitous HTTP protocol—which helps them to be platform-independent—and are easy to organize.

*Coding a Web application client on z/OS doesn't have to be a daunting task.*

The z/OS® platform has for years been labeled "the server of servers" and houses much of

- **IBM Systems Magazine (January / February 2016)**
  - z/OS Client Web Enablement Toolkit Enhances Web Application Availability (pg 34 -36 in hardcopy edition)



# REST easy on z/OS

*Introducing the z/OS Client Web Enablement Toolkit*

**BY STEVE WARREN**

The number of web service applications on the internet has increased significantly in recent years. RESTful applications that use HTTP or HTTPS as a means of communication and send JSON or XML data is as common as it gets in the mobile, client/server world.

Wouldn't it be cool if your existing z/OS applications running in a traditional environment could easily ramp up to play in the game as well as through a set of base z/OS services available to most programs on z/OS?

If you're excited about these options, welcome to the new z/OS Client Web Enablement Toolkit! Built into the base of the z/OS operating system, the toolkit provides a lightweight solution to enable these applications to more easily participate in this client/server space by providing the following built-in features:

- A z/OS JSON parser that can be used to parse JSON text that comes from any source and create new JSON text or add to existing JSON text.
- A z/OS HTTP/HTTPS protocol enabler which uses interfaces similar to other industry-standard APIs.

Just about all environments on z/OS can avail themselves of these new services. Traditional z/OS programs that run in native z/OS have little or no options that they can easily use to participate in web services applications. Programs running as a batch job, as a started procedure or in almost any address space on a z/OS system now have APIs that can be used in a similar manner to any standard z/OS APIs provided by the operating system.

Furthermore, programs can use these APIs in the programming language of their choice. You can use C/C++, COBOL, PL/I, and Assembler languages, and samples are provided for C/C++, COBOL, and PL/I.

Would you like to hear more about the parts of the toolkit and get a small taste of what you can do?

**z/OS JSON parser**

Suppose that you would like to be able to make sense of a large JSON text file that was sent to you from a web server that you are communicating with. The new z/OS JSON parser can do the heavy lifting for you.

The following questions can help you decide which style of parsing is best for you:

Welcome to the new z/OS Client Web Enablement Toolkit!

- Do you know the format of the data that is being returned?
- Are you looking for specific fields in a particular format?
- Do you need to learn about all the data that is returned?

Based on your answer, you can choose the "search" style, the "traversal" style, or a combination of both. The "search" style looks for specific key values in the text stream and then finds the values that are associated with those key values. The "traversal" style starts the traversal parser services to recursively move through the text stream until it learns what was sent.

In either case, a program that uses the JSON parsing services follows this format:

1. Start the JSON parser initialize service (HWTJINIT) to create a parsing instance.
   **Tip:** You can go wild here and create as many parser instances as your application requires. Each parser instance allows the z/OS JSON parser the ability to separately manage the parsing of a JSON text stream. The more instances you have, the more concurrent JSON text streams you can parse.
2. Call the JSON Parse service (HWTJPARS) to have the parser validate the syntax of the text stream and create an internal representation of the JSON text data. Once the data is parsed, it allows all subsequent services to run faster

- **z/OS Hot Topics magazine (August 2015)**
  - **REST easy on z/OS – Introducing the z/OS Client Web Enablement Toolkit** (pg. 26-27)

# Questions?