Contents

Introduction

Welcome to the Mainframe Cloud (MfC) RunTime Adaptor VSAM Reference Guide.

The MfC RunTime Adaptor is a Web Service Agent (WS Agent) that allows web app developers to build applications for the mainframe in web languages such as JavaScript and HTML5.

Intended Audience

This guide is intended for technical personnel who want a detailed overview of the RunTime Adaptor VSAM web service specification, including web service request and response structures and details about web service calls.

Supported Web Services

This guide details the following web services:

- LOGON
- LOGOFF
- VSAM

Installation and Administration

For information regarding installation of the MfC WS Agent refer to the MfC RunTime Adaptor Installation and Administration Guide on the Mainframe Cloud User Resources page.

Web Services

The MfC RunTime Adaptor provides a web service endpoint and runs as a standalone agent on a z/OS mainframe. Once installed the MfC Web Service Agent (WS Agent) is available 24/7. Any standard web application can connect with the WS Agent using a standard web service request.

Web Services Request Structure

A web service request is specified as the path in the URL. The format of requests and responses is standardised across all web services. Requests can be issued in either GET or POST formats. For GET requests, parameters are added to the URL. For POST requests, parameters are specified in a JSON structure in the request body.

GET & POST Request Structure

```
GET url:port/logon
GET url:port/logoff?session=n
GET url:port/vsam?cmd=open&dsn=my.vsam.dataset&mode=rb,type=record
```

```
POST url:port/logon
   No request body required for logon. Logon implies no existing session.

POST url:port/logoff
{ “session”:n }

POST url:port/vsam
{ “cmd”: “open”, “session”: n, “dsn”: “my.vsam.dataset”, “mode”: “rb,type=record” }
```

Examples of the structure of GET and POST requests are listed below.

GET & POST Parameters

Using these examples from the table above:

GET url:port/vsam?cmd=open&dsn=my.vsam.dataset&mode=rb,type=record

POST url:port/sql
{ “cmd”: “open”, “session”: n, “dsn”: “my.vsam.dataset”, “mode”: “rb,type=record” }

The first parameter is the action command.

- For VSAM, it is “cmd”

For GET requests, the first parameter begins after the question mark directly after the web service name, for example, ‘GET url:port/vsam?cmd=open’

Subsequent parameters on a GET URL are delimited by an ampersand character '&'.

For POST requests, the action is specified as a JSON tag, for example, "cmd":"open"

Parameters required by the web service are stated as follows:

- GET URL format: parm1=value1&parm2=value2 etc.
- POST URL format: "parm1":value1, "parm2":value2 etc.

Note that:

- JSON syntax requires that any non-numeric values are specified as strings, enclosed in double quotes.
- Numeric values are not enclosed in double quotes.
- JSON tags can be specified in any order, so for example:

  { "cmd":"open", "session":n } is identical to

  { "session":n, "cmd":"open" }

The session identifier is passed as a parameter:

- GET URL format: session=n
- POST URL format: "session":value

Note that:

- The session identifier is always an integer value. In JSON, it can be specified either in quotes or not in quotes.
- The session identifier is always returned without quotes in the response JSON structure.
- It is accepted in either format in the request structure.

To terminate a socket session

It is possible for the web application to request to terminate a socket session after a request is satisfied. Normally, this would not be generally used. Modern browsers make use of the HTTP V1.1 protocol. The default for HTTP V1.1 is to maintain a persistent socket connection to a target host. Subsequent web service calls to the same target address and port will tend to reuse the same socket connection as a previous request. There is no guarantee that the same socket connection is reused, but often, for the same target address and port, it will be.

If a web client application wishes to force that the current socket connection be terminated after the request is completed, then including a connection close parameter in the GET parameters or the POST request JSON will instruct the agent to terminate the socket after

the response is sent. Alternatively, a Connection header with the value of close can be used. Browsers tend to prevent the user specification of the Connection header but a non-browser-based web application can supply a Connection header. A close connection request will only close the socket connection. It will ***not close*** any active VSAM files nor will it terminate the web service application task associated with the VSAM session. For example:

GET url:port/vsam?connection=close&cmd=commit&session=2

or

POST url:port/vsam
{ “connection”:“close”, “cmd”:“commit”, “session”:2 }

Web Services Response Structure

The response to a web service request is a JSON structure.

```
{  “rc”: http_status_code,
   “agentVersion”:“version number”,
   “session”:n,
   “message”:[ “message 1”,  “message n”   ],
   “vsamresp”:{vsam-web-service-response}
}
```

The last entry (vsamresp tag in the example above) is specific to the actual web service called.

- VSAM requests are specified as: “cmd”:“vsam_command”
- VSAM response returned is: “vsamresp”:{JSON structure specific to VSAM service}

The other fields in the JSON response structure are common to all web service calls.

rc	The HTTP status code that is returned.
agentVersion	Identifies the version of the MfC WS Agent. This should be advised when making support calls to Mainframe Cloud. The agent version is also shown in the MfC WS Agent job log.
session	The session identifier that is assigned to this application thread. This is returned on every response where a valid active session has been assigned. This session identifier must be specified on subsequent requests that are intended to be part of the same logical session. The session identifier is always a whole integer, and the value is returned as an integer not enclosed in double quotes.

message Any error or informational messages pertaining to the general request are returned as an array of message texts. Typically, any messages from security verification are returned here. Any invalid use of session identifiers is also reported here.

Session Identifier Management

A standard RESTful web service operates as a standalone request-response structure. In general, there is no context saved between one call and the next. In the case of the MfC WS Agent, it is necessary to maintain a persistent subtask environment for any opened VSAM files, so it can take advantage of several VSAM requests making up a single unit of work.

The MfC WS Agent maintains 2 separate subtasks/threads for each request/response transaction. One thread is the logical session with the application, for example an application session with opened VSAM files, and the other thread is the associated sockets task where the request was received. The dropping of a socket connection will not affect the existence of the application session. The agent assigns a connection identifier with the socket connection and a session identifier with a logical application session such as VSAM.

A browser may or may not choose to issue subsequent requests on the same socket connection as the previous request. A session identifier is returned in the response for every request (unless the logical session is terminated). If the intent is to send a subsequent request to the same logical session as a previous request, the client web application must specify this session identifier on the next transaction request. The browser has 3 choices of which socket connection it chooses to send the request:

1. The same socket connection as the previous request.

2. Establish a brand-new socket connection.

3. An existing socket connection that was last used by a different client application thread talking to a different logical session with a different session identifier (and possibly a different userid). Existing active (non-busy) socket connections to the same target IP address and port (i.e. same MfC WS Agent) can be chosen. This choice can also apply to different tabs on the same browser window. ‘Non-busy’ is defined as an active socket connection that is not waiting for a response from a previous send request.

In the first case, the request is simply passed onto the correct application thread with a matching session identifier, without any further possible switch checking.

In both the second and third cases, the agent will determine the intended target application thread by the specified session identifier. The request will then be transferred to the correct thread.

The user need not be concerned as to which choice of socket connection is used by a browser. As long as a session identifier is specified on a request, the request will be directed to the appropriate session. The session identifier is unique across the lifetime of the agent region. Access to an existing session identifier thread is only allowed if the request userid is the same as the userid that created the application session in the first place. If a different userid is specified, then the specified session identifier will not be found and a 404 Not Found HTTP status will be returned.

It is possible to have several socket connection threads all pointing to the same application session. If so and they all receive new requests, they will be directed to the same application session in turn.

If a session identifier is not specified on a particular request, a new application thread will be started with its own new session identifier value. This request can come in on any socket connection thread, i.e. any of the 3 cases above. In all cases, the request will be directed to a brand-new application thread and a new session identifier assigned.

A client web application may open multiple VSAM files on a given logical session. Each opened VSAM file will be assigned a file handle identifier. Subsequent requests to a specific VSAM file must specify the file handle that was returned from the open request.

The logoff request will terminate the associated socket connection (that the logoff request was received on), rollback any pending updates (if applicable), close all opened VSAM files on the target session, and finally terminate the application task session. A VSAM close on one or all VSAM files does not terminate the application task nor the socket session.

Escaping

There are two forms of escaping involved with the MfC WS Agent.

Parameters on a GET URL can be escaped with a %xx construct. The xx is the ASCII code for the required character. Browsers tend to do this automatically, but there is no reason why a web application cannot also do this itself. The WS Agent will de-escape parameters in the URL as applicable.

The other form of escaping is the standard JSON escaping sequences. The escaping syntax is well documented on the web. Request JSON created by the web application must be escaped as required to send valid JSON data.

The main commonly used requirement is to escape the double quote and backslash characters (“ and \). Any character can be escaped, but these two are the common required ones. There are also special escape cases for \u, \b, \t, \n, \f and \r.

Double quotes are used extensively in JSON structures and backslash is the escaping precedent character. Any use of these 2 characters as part of data must be escaped by preceding it with a backslash character (i.e. \" and \\).

All response JSON output sent by the WS Agent will be suitably escaped. So, a web application would need to de-escape the response as required.

Security

The mainframe security credentials are userid, password and optional new password. All are maximum 8 characters.

If the Security Interface Module (SIM) is active, full security authentication is performed. If a security check fails, the web service will not be actioned, and an appropriate message will be returned.

Example – Authenticate WITH new password

```
// Sample TSO login variables.
var userid   = "TSOUSER";     // Your TSO USERID.
var pswd     = "TSOPASS";     // Your TSO PASSWORD.
var newpswd  = "NEWPASS";     // Your NEW TSO PASSWORD.
// Set Authentication variable.
var lgnpwd = userid + ":" + pswd + ":" + newpswd;
// Authentication on the HTTP Request Header.
xhttp.setRequestHeader( "Authorization", "Basic " + btoa( lgnpwd ) );
```

Example – Authenticate WITHOUT new password

```
// Sample TSO login variables.
var userid   = "TSOUSER";     // Your TSO USERID.
var pswd     = "TSOPASS";     // Your TSO PASSWORD.
var newpswd  = "";            // *NO* NEW TSO PASSWORD.
// Set Authentication variable. Either of the following is acceptable.
var lgnpwd = userid + ":" + pswd + ":" + newpswd;
var lgnpwd = userid + ":" + pswd;
// Authentication on the HTTP Request Header.
xhttp.setRequestHeader( "Authorization", "Basic " + btoa( lgnpwd ) );
```

In the event of a password expired, a new password must be supplied on the next request. The password can be changed on any request. If a new password is supplied, the existing password must also be supplied. The existing password will be validated before the password is changed to the new value.

Password Token

Userid, password and new password credentials are passed in an encoded format via the Authorization request header, for example:

Authorization: Basic B7Xjzfi6cAUPu19xb676

On the first validation of a password, an authentication token is returned to the caller. This token is returned via the **Authentication-Info** response header, for example:

Authentication-Info: auth="YeOSOqoK"

This token should be used as the password for subsequent web service calls. This token is only applicable to the same userid and IP address of the initial call. It will expire after a period of inactivity. The initial validation request and all subsequent requests using the same token constitute a single logon session, for accounting purposes.

Example – Get Authentication Token from Response Header

```
xhttp.onreadystatechange = function() {
  if ( this.readyState == 4 && this.status == 200 ) {
    // Response Header Auth Token.
    var authin = xhttp.getResponseHeader( "Authentication-Info" );
    // Use returned Token for subsequent web service calls.
    if ( authin != null ) pswd = authin.substr( 6, 8 );
  }
};
```

LOGON and LOGOFF

The structure of GET and POST requests for LOGON are:

GET url:port/logon

or

POST url:port/logon

This will simply validate the logon credentials and create an active session. There is generally no need to issue this request, as the first request will do an implicit logon.

The structure of GET and POST requests for LOGOFF are:

GET url:port/logoff?session=n

or

POST url:port/logoff
{“session”=n}

This will terminate the nominated session. There can be multiple active web services open for a particular session. For example, there may be a connection to Db2, open VSAM files and opened IMS PSBs. A LOGOFF request will be sent to each of the active web services, which will perform their own termination logic to any opened files. If there were any pending updates on any of the web services, then an internal ROLLBACK will roll back any and all pending updates on all active web services under that session.

A logoff request should be issued as the final request, to clean up and terminate the current session gracefully. If the application thread is not shutdown, it will eventually timeout due to inactivity and then it will close down passing an internal LOGOFF request to the session.

It should be noted that terminate type requests to a specific web service (e.g. VSAM CLOSEALL or IMS TALL) do not terminate a session. A logoff should always be issued as the final request to properly shutdown a session.

VSAM Web Service

The VSAM web service caters for read, locate, update, insert and delete of records for VSAM KSDS, ESDS and AIX files. RRDS is not supported at this stage, though it may be added in a future release. VSAM linear files are not supported.

The structure of GET and POST requests for VSAM are:

GET url:port/vsam?cmd=command&parm1=value1&parm2=value2 ...

or

POST url:port/vsam
{ “cmd”:”command”, “parm1”:“value1”, “parm2”:“value2 .... }

If an existing logical session is to be used, then add **“session”:n** as applicable.

If an existing VSAM file handle is to be used, then add **“fh”:n** as applicable.

The body of a typical sequence of requests may be:

- { “cmd”:“open”, “dsn”:“my.vsam.dataset”, “mode”:“rb,type=record” }
- { “session”:1, “cmd”:“read”, “fh”:1, “key”:“key_value” }
- { “session”:1, “cmd”:“read”, “fh”:1, “count”:3 }
- { “session”:1, “cmd”:“update”, “fh”:1, “data”:“record_data” }
- { “session”:1, “cmd”:“close”, “fh”:1 }

And a final request to logoff the session using the logoff web service:

- GET url:port/logoff?session=1

If operating in TVS mode, then commit and/or rollback commands need to be included to commit/backout updates.

Request JSON Structure

All HTTP POST requests need to supply a JSON request structure. Each VSAM action command will have different parameters. These are described under the relevant command.

```
{ "cmd":"vsam_cmd", "session":3, "parmx":valuex, ... }
```

Response JSON Structures

The VSAM web service returns a “vsamresp”:{JSON_structure}. This is inserted into the outer JSON structure described earlier in this document.

The various action command response attributes are described under the relevant VSAM command descriptions below.

Copybook Response Format

Records of a VSAM dataset are plain records. There is no subdivision into fields, similar to how Db2 row entries are split into separate fields. Typically, VSAM applications read/update whole records as a single string of data. The programs interpret the record content into a group of fields internally. For Cobol programs, this definition of fields within a record is commonly set up as a so-called copybook definition. The structure layout of a record is defined as standard Cobol data definitions and this layout is saved in a separate copybook file. The Cobol program is then compiled which includes the required copybook definitions.

A simple copybook map may look like this:

```
01 LINE-DATA-1.
    05 TRAN-DESC-TERM                PIC X(10).
    05 TRAN-DATE                     PIC X(10).
    05 TRAN-GST           COMP-3     PIC S9(11)V99.
    05 TRAN-AMT           COMP-3     PIC S9(11)V99.
```

A record of the VSAM file could be:

```
Balance   2019-01-18...........
```

For a plain read request, that is the data line that would be returned to the web application.

The data in columns 21 through 32 would be in packed decimal format and therefore not readable characters. As the data is returned in JSON format, the data string would be escaped, so it will, for example, include strings like \u0000\u0012 etc.

A RunTime Adaptor VSAM request to read this record can return this data in ASCII, EBCDIC, HEX, BASE64 or a copybook map format.

For a VSAM read specifying the LINE-DATA-1 copybook map as listed above, the MfC Runtime Adaptor VSAM interface will return this record like this:

```
{ "LINE-DATA-1" : {
    "TRAN-DESC-TERM": "Balance",
    "TRAN-DATE": "2019-01-18",
    "TRAN-GST": 123.45,
    "TRAN-AMT": 12345.00
  }
}
```

The numeric items are converted to a display format suitable for processing by a web application. Any string items (i.e. those governed by a PIC X declaration) are escaped where necessary.

Supported Copybook Features

The Cobol structured layouts support a number of features. Most of the common features will be supported by the MfC copybook interface. Other lesser known features will be added in future releases.

In the example above, only 2 levels were used – 01 and 05. The 05 fields are a subset of level 01 field. The 05 level fields can be further divided down to further levels. There is no limit to the number of sub-level structures supported (there actually would be a limit given the fact that the level number is 2 digits long).

Cobol allows the same named field within different hierarchy structures. This is supported in the MfC interface. But the MfC agent referencing copybook entries that refer to other fields do not take possible duplicate names into account. With the initial implementation, this only applies to the field name specified in the copybook map name and SHOWIF statements. A search of the specified field name will match on the first found occurrence of that field name.

Usage values of DISPLAY and the numeric types of COMP and COMP-1 through COMP-5 are all supported. Usage NATIONAL is not supported.

Picture values of 9, A, X, P, V and S are supported.

Sign leading and separate character options are not currently supported. Specification of these is tolerated, but otherwise ignored. These would be rarely used features.

The multitude of edited picture values is not supported. These would not normally be used to describe record layouts. They are more typically used to describe formatted print output layouts, whereas the typical intention of copybook format in an MfC environment would be to read raw files which would normally not include edited picture formats.

Fixed size arrays are supported, such as used by the 'OCCURS n TIMES' clause, where n is a specific integer. The variable size arrays as used by the 'OCCURS n TIMES DEPENDING ON field name' clause is not supported in the initial implementation.

REDEFINES is supported. With the initial implementation, all REDEFINES structures will be formatted into field name/value pairs. In a typical application, the intent would likely be to use a specific REDEFINES structure based on the content of another field and to bypass all other REDEFINES of the same structure. Such conditional logic is embedded in a Cobol program; there is no specific mention as such in the copybook structure. SHOWIF support does allow the expansion of specific REDEFINES structures based on the value of a separate field – refer to the SHOWIF feature described below.

The REDEFINES clause references a previously defined field at the same hierarchy level. The current logic locates the first occurrence of the named field. If there are no duplicate entries for the same name (duplicate field names are allowable if they are at different hierarchy levels), then the search and locate will work as expected. If there are duplicate entries for the named field, then the search and locate will return the first found entry. This may not be the expected result. Good programming practice should dictate that duplicate names are not used in this scenario.

Although some error validation is performed on input copybook map source, the error validation performed by the MfC copybook interface is not intended to be 100% complete. It is expected that the copybook source is already used by Cobol programs and thus should be valid syntactically correct structures in their own right.

Only minimal validation of numeric fields is performed. Packed decimal fields are validated to be proper packed decimal format – invalid packed decimal values are reported as '*'. There is no validation done on floating point values. There is no validation done on numeric usage display fields. The value contained will be output as is. If it is not numeric, then the web application will see a non-numeric returned. Such usage numeric display fields will be returned as strings normally enclosed in double quotes as standard.

If COMP-4 fields contain values that violate their picture constraints, the MfC routines will treat them as COMP-5 fields for the purposes of interpreting the value. A well-behaved Cobol program should not save values in COMP-4 fields that violate their picture constraints. Our understanding is that a Cobol program truncates the values as necessary to satisfy the picture constraints. However, certain uses of compiler options, REDEFINES usage and the like can cause this check to be bypassed.

The SYNC option is tolerated, but otherwise ignored. No attempt has been made to add padding bytes to satisfy the SYNC option. For the initial release, any required padding bytes need to be explicitly added to the copybook source. Although not required, this should be done for good programming practice. Cobol does add padding bytes where necessary to satisfy synchronisation on appropriate half-word, full-word or double-word boundaries.

Copybook Metadata

Copybook metadata can be requested. Metadata provides a description of the copybook formatted data. The syntax of the returned metadata is –

```
"datatype": { <<entry>> }

Where <<entry>> is defined as follows:

<<entry>>
  "name": {
      "TYPE": "INT",          /* not for group entries */
      "WIDTH": 2,             /* not for group entries */
      "REDEFINES": "name",    /* if applicable */
      "GROUP": {              /* Only for group entries */
          <<entry>>,          /* recursive entry structure */
          <<entry>>,          /* recursive entry structure */
          Etc...
      }
  }
```

The possible types returned are:

CHAR, SHORTINT, INT, LOGNINT, FLOAT, DOUBLE and DECIMAL

The width value is the maximum display width of the field.

The following copybook map –

```
05 RECORD.
       10 YEAR    PIC 9(4).
```

will result in this metadata returned –

```
"datatype":{
    "RECORD":{
        "GROUP":{
            "YEAR":{
                "TYPE": "SHORTINT",
                "WIDTH": 4
            }
        }
    }
}
```

A VSAM read record requesting copybook format can include the tag “datatype”:“yes” in the request JSON. This will result in the datatype metadata being returned in the response JSON.

SHOWIF

SHOWIF support adds the capability to selectively format copybook entries based on the content of a named field. A practical use of this is the ability to selectively format a REDEFINES section. An example will demonstrate this feature.

```
01 EMP-REC.
   05 EMPNO                          PIC 9(6).
   05 EMP-TYPE                       PIC X.
   05 MANAGER-REC.
*SHOWIF EMP-TYPE='M'
      10 MGR-SALARY                  PIC 9(7)V99.
      10 MGR-DETAILS                 PIC X(30).
   05 WORKER-REC REDEFINES MANAGER-REC.
*SHOWIF EMP-TYPE='W'
      10 WRK-SALARY                  PIC 9(5)V99.
      10 WRK-DETAILS                 PIC X(31).
```

The *SHOWIF statements have an asterisk in column 1; they appear as a comment, but are referenced by the MfC copybook interpreter. The SHOWIF referenced field must be defined before it is referenced in a SHOWIF statement. The referenced field should not be a duplicate named field; if so, then the locate of the referenced field will find the first reference which may or may not be the intended target field.

In the above example, either the MANAGER-REC or WORKER-REC sections will be formatted, depending on the value of the EMP-TYPE field in each record. The length of the SHOWIF comparison value can be less than or equal to the length of the referenced field. If the comparison value length is greater than the length of the referenced field, the statement will be rejected.

RLS and TVS

RLS and TVS are IBM features related to VSAM Record Level Sharing.

Without RLS and TVS, sharing controls of VSAM datasets is handled by the SHROPTIONS value that was specified when a VSAM dataset was initially defined. Depending on the SHROPTIONS value, system integrity is handled at the file level and/or the control interval level.

Applications tend to add their own sharing/integrity controls on top of this. An ENQ/DEQ facility has been included in the VSAM web service to accommodate this.

RLS (Record Level Sharing) is an IBM feature that takes most of the sharing and integrity controls to the system level. SHROPTIONS are not used. All users can read and write (subject to standard security access checks) and integrity is handled at the record level. This greatly simplifies VSAM file sharing.

TVS is an extension to RLS. It offers more sharing and integrity capabilities using sysplex capabilities.

The MfC RunTime Adaptor VSAM web service supports both RLS and TVS. When TVS is enabled, the 2-phase commit protocol is used. The web application will need to issue specific commit (or rollback) commands to effect permanent updates.

VSAM Commands

This section describes the syntax of the inbound JSON structure and outbound JSON response structure for each VSAM action command. The command itself and an optional sessionid is passed with every request. Only the JSON request structures that are sent with POST requests are shown here. The equivalent GET request with the parameters on the URL can be specified if desired. Though it is recommended that POST be used as specifying parameters in the URL can exceed the maximum length of a URL. All keywords in the JSON request structure are not case sensitive; it is typical for most JSON constructs to lower case keyword names. Values that depict a certain value, such as SCOPE that has one of 3 valid values - these are not case sensitive; they can be entered in any case. Values that are variable are typically case sensitive in that they are passed to the relevant z/OS service as is. The service may then uppercase them to its own standards or it may not (e.g. dataset names are uppercased by the system regardless of how they were specified). Users should consult relevant IBM documentation as required to determine case issues.

Open

Request

```
{
  "cmd":char,      Command = OPEN
  "dsn":char,      VSAM Dataset name – cluster or path
  "ddname":char,   Dynamic or JCL allocated ddname
  "mode":char      Mode
  "qname":char     ENQ qname
  "rname":char     ENQ rname
  "scope":char     Scope – SYSTEMS, SYSTEM or STEP
}
```

Dataset name is the fully qualified dataset name of the target VSAM file.

Mode options – these are passed as is to the z/OS VSAM API, as documented in the z/OS C runtime library manual.

- rb open for read only, file must not be empty
- wb open for write only, file emptied on open
- ab open for update only, file not emptied on open

Addition of + to the above opens for read and write: e.g. rb+, wb+, ab+

- type=record All VSAM I/O requests are record type, must be specified.
- acc=value, Access direction, default = fwd
    - fwd Open with access direction foreward. Filepos at open will point to first record. Default.
    - bwd Open with access direction backward. Filepos at open will point to last record.

- pswd=pswd Optional password for a VSAM dataset
- rls=value VSAM RLS/TVS mode (nri) No Read Integrity, (cr) Consistent Read, (cre) Consistent Read Explicit. Cre selects TVS mode, nri and cr select RLS mode.

qname, rname and scope are parameters as passed to the z/OS ENQ and DEQ assembler macros. If ENQ serialisation is required for this file, then these need to be specified on the open request. Scope can be SYSTEMS, SYSTEM or STEP. Refer to z/OS documentation for more information on ENQ/DEQ.

Mode = rb allocates with DISP=SHR, wb and ab allocates with DISP=OLD

Use of RLS/TVS requires the setup of the SYSVSAM server and sysplex requirements on the system.

Returns an integer file handle (fh) which is required on all subsequent VSAMIO calls. The handle is unique across the agent. It identifies this particular opened file in this session.

The above JSON request string is the content of HTTP body for a POST request. For GET requests, these are specified as parameters on the URL:

e.g. http:host:port/vsam?cmd=open&dsname=vsam.file.name&mode=rb+,type=record

Response

```
{"cmd":char,                 Command = OPEN
 "fh":integer,               File handle
 "dsn":char,                 Dataset name
 "vsamtype":char,            KSDS|ESDS|RRDS|KSDS_PATH|ESDS_PATH|
                             NOT_VSAM
 "modeflag":char,            mode flag used at open – e.g. rb+
 "acc":char,                 Processing direction – FWD(forward)
                             or   BWD(backward)
 "rls":char,                 NORLS | RLS | TVS
 "maxreclen":integer,        Maximum record length
 "keypos":integer,           Key position offset within record.
                             First byte is offset zero.
 "keylen”:integer,           Key length
 "enq":char,                 Enqueue lock held – yes or no
 "qname":char,               Enqueue qname from open request
 "rname":char,               Enqueue rname from open request
 "scope":char,               Enqueue scope from open request
 "message":["string",…]      Return messages
 "rc":integer                Return code
}
```

The cmd value is always echoed back in the response JSON. File handle is always returned (unless the open fails) for all calls (except for CLOSE and LOGOFF). The return code tag and any messages are returned for all request calls. All the other tags are informational type tags on the status of the file. All of these are also returned on a GETINFO call.

Error tags on Error

```
"errno”:integer,        Error number (from open)
"errtext":char,         Text associated with errno
"vsamrc”:integer,       VSAM return code
"vsamrsn":integer,      VSAM reason code
"lastop”:char,          Last VSAM service call. VSAM calls will
                        result in suitable text. Non VSAM values
                        will result in a number code.
"rplfdbk":char          RPL feedback code, 8 hex digits
```

The rc tag is the overall return code. This is returned for all calls:

- RC = 0 indicates success
- RC = 4 indicates informational messages returned but no VSAM error. Read EOF and a locate which found no record will return RC=4.
- RC = 8 indicates error identified by the API. Suitable message(s) will be returned.
- RC = 12 indicates a VSAM error. Messages and other VSAM codes will be returned.

For VSAM errors (RC = 12), other error information is returned in the JSON response.

For errors in open, generally only errno and errtext are returned.

For all other VSAM calls, errno is generally not used, other VSAM codes are returned:

Any returned messages are returned in the message array tag.

Getinfo

Return VSAM information – VSAM type KSDS/ESDS, key position and length, open mode, direction, current key, current RBA, state, etc. This information is also returned on the open call. Issuing a separate getinfo() call will obtain the information again.

Request

```
{
 "fh":integer           File handle
}
```

Response

```
 {
  "cmd":char,           Command = GETINFO
  "fh":integer,         File handle
  "dsn":char,           Dataset name
  "vsamtype":char,      KSDS | ESDS | RRDS | KSDS_PATH |
                        ESDS_PATH | NOT_VSAM
  "modeflag":char,      mode flag used at open – e.g. rb+
  "acc":char,           Processing direction - FWD or BWD
```

```
  "rls":char,           NORLS | RLS | TVS
  "maxreclen":integer,  Maximum record length
  "keypos":integer,     Key position offset within record
  "keylen”:integer,     Key length
  "enq":char,           Enqueue lock held – yes or no
  "qname":char,         Enqueue qname as passed on the open
  "rname":char,         Enqueue rname as passed on the open
  "scope":char,         Enqueue scope as passed on the open
  "rc":integer          Return code
 }
```

Read

Read a previously opened dataset given a file handle. Key (or RBA) may be specified, this will locate to the specified key (or RBA) before the read is done.

Request

```
 {
  "cmd":char,           Command = READ
  "fh":integer,         File handle
  "key":char,           Requested key value (for KSDS or AIX
                        key). If specified, a locate to this key
                        will be performed before the read. Key
                        and rba options are mutually exclusive.
  "rba":int,            Requested RBA key (for ESDS or KSDS). If
                        specified, a locate to this rba will be
                        performed before the read. Key and rba
                        options are mutually exclusive.
  "count":int,          Count of records to read, default = 1
  "keyfmt":char         Key format, either ASCII, EBCDIC, BASE64
                        or HEX. Default = ASCII
  "recfmt":char         Record format, either ASCII, EBCDIC,
                        BASE64 or HEX. Default = ASCII
  "lock":char           getrel, getret, rel
  "share":char          E (EXCL) or S (SHR) for the ENQ that was
                        supplied on the open request.
  "copyb":char          Copybook name specified as
                        member.fieldname
  "datatype":char       YES | NO. Indicates whether to return
                        copybook metadata in the response.
 }
```

Response

```
 {
  "fh":integer,             File handle
  "eof":"yes",              Returned when EOF reached
  "rc":integer,             Return code
  "numrecs":integer,        Count of records returned
  "datatype":char,          Datatype JSON structure
  "data":["record1", ..],   Array of returned records
  "rba":rba                 RBA of last read record
 }
```

Read a previously opened dataset given a file handle.

Read count records starting with a specified key. If key is omitted, read the next record after the previously read record. If key is supplied, a full KEY_EQ search is done to locate the exact key. If other locate options are desired, then use a separate locate service call first.

Data records are returned in the nominated format as indicated by the recfmt tag.

For copybook format requests, the copyb tag specifies a value `member.fieldname`. Where member is a PDS member of the copybook PDS pre-allocated to the web services agent region. Fieldname is the fieldname within that copybook source that is to be mapped to a read record. The returned data records will be in ASCII format and be similar to this format –

```
"data":[
    { "EMP-REC":{
          "EMP-NO":"123456",
           Etc for other fields
      }
    },
    { Entry for second record },
    ...
]
```

Key format indicates the format of the supplied key for a KSDS or an AIX file. The format can be ASCII, EBCDIC, BASE64 or HEX. The default is ASCII. ASCII indicates that the supplied key is in ASCII format. It will be converted to EBCDIC format before it is used for a locate request. A key supplied in EBCDIC format will be the actual value that will be used to locate the record. BASE64 and HEX formats will be decoded to the raw format before being used in a locate request. It should be noted that BASE64 and HEX decoding is final; the input to the original encoding must have been in EBCDIC raw format, not in ASCII format.

For ESDS files, key will be an RBA (Relative Byte Address). The RBA will be in hexadecimal format up to a maximum of 16 hex digits (i.e. 8 bytes). Leading zeroes do not need to be specified. An RBA value will be returned for every read request – this RBA value can be used for a subsequent locate.

An RBA value can be used for KSDS files, but it is not recommended. RBA values for a particular record can change after any update, delete or insert operations. RBA values are always consistent for ESDS records.

The file position pointer is updated with every read operation. Subsequent read requests will read the next record after (or previous record before if current direction processing is backward) the last read request.

A read request cannot directly follow a write request without an intervening reposition request. Locate, rewind and read with a key are considered reposition requests.

A read request cannot directly follow a failed update request without an intervening reposition request.

A read request must immediately precede an update or delete request of the same record.

A lock value of getrel (Get Release) or getret (Get Retain) issues an ENQ prior to the operation. If the ENQ lock is already held, it is treated as a noop. If the ENQ lock is already held with SHR but the current request is EXCL, the request is rejected. If this is desired, then either a separate ENQ CHNG web service call should be issued or alternatively a DEQ followed by an ENQ could also be used. A getret (Get Retain) holds the enqueue active after the request is complete. A getrel or a rel request releases the ENQ after the request is complete.

The RBA of each returned record will be returned. This value can be used for a subsequent locate request. It should be noted that usage of RBAs is not recommended for KSDS. Even though RBA's will work for KSDS, RBAs are subject to be changed after update and delete activity. RBAs are not applicable for use with AIX files – use key instead.

Locate

Locate a record given a key. This sets the file position pointer, a subsequent read will read that record. Direction of sequential access can also be set here.

Request

```
{
 "fh":integer,          File handle
 "key":char,            Requested key value (KSDS or AIX key)
 "rba":int,             Requested RBA key (for ESDS)
 "keyfmt":char          Key format, either ASCII, EBCDIC, BASE64
                        or HEX. Default = ASCII
 "locopt":char          Option value
 "lock":char            getrel, getret, rel
}
```

Response

```
 {
  "cmd":char,           Command = LOCATE
  "fh":integer,         File handle
  "rc":integer          Return code
 }
```

Locopt can be one of –

- KEY_FIRST
- KEY_EQ
- KEY_GE
- RBA_EQ
- KEY_LAST
- KEY_EQ_BWD
- RBA_EQ_BWD

KEY_FIRST and KEY_LAST do not require a key value to be specified. The other KEY_* options require a full or partial key value.
KEY_FIRST, KEY_EQ, KEY_GE, RBA_EQ set the access direction to forward.

KEY_LAST, KEY_EQ_BWD, RBA_EQ_BWD set the access direction to backward.

RBA_EQ and RBA_EQ_BWD are invalid for paths and not recommended for KSDS and RRDS files. They are primarily intended for ESDS files.

KEY_EQ, KEY_GE – a partial of full length key can be specified.

KEY_EQ_BWD – a full key (of the files key length) search must be specified.

Key will be an RBA value for ESDS files. If > 4 bytes, then it is an extended 8 byte RBA value.

Rewind

Reset back to the first record. A locate with KEY_FIRST (for KSDS and paths) is identical to a rewind call. A rewind call does not change the access direction. If the access direction was backward and a rewind call is issued, the file position is reset to the beginning of the file but the access direction will remain as backward. A locate with KEY_FIRST will reset the file position to the beginning of the file and will also change the access direction to forward.

Request

```
{
 "cmd":char,           Command = REWIND
 "fh":integer,         File handle
 "lock":char           getrel, getret, rel
}
```

Response

```
{
 "fh":integer,         File handle
 "rc":integer          Return code
}
```

Write

Insert a new record. For KSDS, the key dictates where the record will be inserted. ESDS writes always occur after the last record.

Request

```
{
 "fh":integer,         File handle
 "recfmt":char,        Record format, either ASCII, EBCDIC,
                       BASE64 or HEX
 "lock":char,          getrel, getret, rel
 "record": "record"    Record to be written
}
```

Response

```
{
 "cmd":char,           Command = WRITE
 "fh":integer,         File handle
 "rc":integer,         Return code
 "rba":rba             Returned RBA (ESDS only)
}
```

Update

Update an existing record.

Request

```
{
 "fh":integer,         File handle
 "key":char,           Current file position, a key value
 "rba":int,            Current file position, RBA (Relative
                       Byte Address) for ESDS
 "recfmt":char,        Record format, either ASCII, EBCDIC,
                       BASE64 or HEX
 "keyfmt":char,        Key format, either ASCII, EBCDIC, BASE64
                       or HEX
 "record":"record",    Record to be updated
 "verify":char,        yes, no
 "lock":char           getrel, getret, rel
}
```

Response

```
{
 "fh":integer,         File handle
 "rc":integer,         Return code
 "rba":rba             Returned RBA (ESDS only)
}
```

Before an update is issued, the record must be previously read by the preceding request. A copy of the record from the last read is retained by the agent, provided that there has been no intervening other request since the last read for this file. The supplied key must match the key in the supplied record and must also match the key of the previous read request.

If verify parameter is set to yes, then the agent will verify the record before the update is performed. The record will be re-positioned to and re-read and the contents compared to the previous read buffer (this is retained from a previous call). Only if the record still exists and the contents match will the update be done.

The agent will determine if the key itself has been updated. If the key has been updated, then the update would effectively become a delete of one record and an insertion of a different record. Updates where the key is modified are not allowed by the API. If this is desired, the web application should delete the existing record and insert a new record.

For a keyed file, the key is obtained from the record. This will be checked and matched to the last read record. If no match or there is no last read record, then the update will be rejected. For an ESDS file, the associated RBA of the record being updated must be supplied. This will be checked and matched against the last read record. If there is no match, the update will be rejected.

Delete

Delete a record, given a specified key. This record must be previously read without any other intervening VSAMIO operation so the current file position is already positioned at the target record.

Request

```
{
 "fh":integer,          File handle
 "key":char,            Current file position, a key value
 "rba":int,             Current file position, RBA (Relative
                        Byte Address)
 "verify":char,         yes, no
 "lock":char            getrel, getret, rel
}
```

Response

```
{
 "cmd":char,            Command = DELETE
 "fh":integer,          File handle
 "rc":integer           Return code
}
```

Before a delete is issued, the record must be previously read by the preceding request. A copy of the record from the last read is retained by the agent, provided that there has been no intervening other request since the last read for this file. The supplied key must match the key of the previous read request.

If verify parameter is set to yes, then the agent will verify the record before the delete is performed. The record will be re-positioned to and re-read and the contents compared to the previous read buffer (this is retained from a previous call). Only if the records still exists and the contents match will the delete be done.

For a keyed file, the key is obtained from the record. This will be checked and matched to the last read record. If no match or there is no last read record, then the delete will be rejected. For an ESDS file, the associated RBA of the record being deleted must be supplied. This will be checked and matched against the last read record. If there is no match, the delete will be rejected.

Commit

Commit and rollback apply for when using commit protocol, that is when rls=cre is specified on the mode parameter of the open request.

A commit request commits any outstanding changes permanently.

Request

```
{
 "fh":integer                File handle
}
```

Response

```
{
 "cmd":char,                 Command = COMMIT
 "rc":integer                Return code
}
```

Rollback

Commit and rollback apply for when using commit protocol, that is when rls=cre is specified on the mode parameter of the open request.

A rollback request rolls back any pending changes since the last commit/rollback/open request.

Request

```
{
  "fh":integer               File handle
}
```

Response

```
{
 "cmd":char,                 Command = ROLLBACK
 "rc":integer                Return code
}
```

Close

Close a file. Any outstanding ENQ for this file will be released.

Request

```
{
  "fh":integer               File handle
}
```

Response

```
{
 "cmd":char,                 Command = CLOSE
 "rc":integer                Return code
}
```

CloseAll

Terminate all VSAM access for this userid session. Close all opened files.

The LOGOFF web service implicitly initiates a CloseAll call. As such, if the intent is to completely logoff a user session, a LOGOFF web service call will terminate all active web services for that user session. This includes issuing an implicit Close All call to the VSAM web service code. In a similar vein, an agent region shutdown will issue a LOGOFF web service call to all active web service subtasks. This in turn will issue a CloseAll request to close any opened files.

Request

```
There are no request parameters.
```

Response

```
{
 "cmd":char,            Command = CLOSEALL
 "rc":integer           Return code
}
```

Logoff

A Logoff call is similar to CloseAll in that it closes all VSAM opened files and terminates all VSAM access for this userid session. A Logoff call is generally implicitly issued when a session termination or region shutdown is detected. A user-initiated session termination can occur by the user specifically calling the LOGOFF web service for a particular session. A session timeout on an inactive session will also cause a session termination.

A Logoff command issued to the VSAM web service will be treated the same as a CloseAll command call. To avoid confusion, Web applications should use Close and CloseAll to close VSAM files, and the LOGOFF web service to terminate a session.

If a web application wishes to terminate all activity for their session, then a LOGOFF web service should be used. This will ensure that all activity related to that user session will be called with a Logoff request call. This LOGOFF web service will implicitly issue Logoff requests to all open web services pertaining to that user session, not just VSAM files.

Request

```
There are no request parameters.
```

Response

```
{
 "cmd":char,            Command = LOGOFF
 "rc":integer           Return code
}
```

Diagnostics

SHOW Command

SHOW TASK

Show all tasks

```
SHOW TASK
M00210 Task Type     User     Session
M00211 0001 MAIN
M00211 0002 LOG
M00211 0003 TCP_LSTN
M00211 0004 CEATSO
M00211 0006 APP2     ADCDT    2
M00211 0007 APP1     ADCDT    2
M00211 0008 APP1
M00211 0014 APP1
M00201 *End*
```

The APP* tasks are applicable to user sessions.

The first 4 tasks are system tasks. Task numbers are assigned sequentially.

Any task that has terminated or is unassigned and waiting to be reassigned will not be shown, hence the possible 'missing' task numbers in the display.

SHOW TASK=n

Show a specific task by taskid.

SHOW USER=userid SESS=sessionid DETAIL

SHOW USER and SHOW SESS are identical. The USER and SESS keywords offer the ability to filter by a userid or a sessionid. The DETAIL option provides more detailed information, being the M00214-217 message lines as shown in the display example below.

```
SHOW USER DETAIL
M00212 Task Type     User     Session  Opens
M00213 0014 APP1     ADCDT    3
M00213 0016 APP2     ADCDT    3         ENQ=1 VSAM=2
M00214  VSAM   Mode RLS ENQ   DSN
M00215  KSDS   rb   NO  SHR   MSP.VSAM.KSDS1
M00215  ESDS   rb   NO  SHR   MSP.VSAM.ESDS1
M00216  SHR STAT SCOPE   QNAME    RNAME
M00217  SHR OWN  SYSTEMS MFCENQ   MFC KSDS1
M00213 0018 APP1     ADCDS    4
M00213 0019 APP2     ADCDS    4         ENQ=1 VSAM=2
M00214  VSAM   Mode RLS ENQ   DSN
M00215  KSDS   rb   NO  SHR   MSP.VSAM.KSDS1
M00215  ESDS   rb   NO  SHR   MSP.VSAM.ESDS1
M00216  SHR STAT SCOPE   QNAME    RNAME
M00217  SHR OWN  SYSTEMS MFCENQ   MFC KSDS1
M00201 *End*
```

From the above example, userid ADCDT has 1 sessionid=3 and 2 active subtasks – task 14 and task 16.

The APP2 task has 1 active enqueue and 2 active VSAM opens.

The M00214/M00215 lines show a VSAM KSDS and an ESDS file opened in RB mode with no RLS. ENQ column notes a SHR enqueue held for this file.

The M00216/M00217 lines shows one owned enqueue with the nominated QNAME, RNAME and scope. The following lines show a similar story with userid ADCDS and sessionid=4.

KILL Command

KILL TASK=n SESS=sessionid FORCE

From a SHOW display, specific task numbers and sessionids can be obtained. A kill command can terminate a specific session or a specific task. Only application tasks can be terminated by the kill command. A kill command will implicitly issue an internal Logoff command to the task. If the task is currently busy, the logoff will be delayed until the task completes its current processing and becomes idle again. A task can show as busy when it is held up by a waiting enqueue.

In the event that a task is busy, or it may be waiting for an enqueue, a Kill command will not take effect until the task becomes idle again. If it remains busy and a kill result is wanted, then the FORCE option can be specified. The force option will attempt an implicit logoff first. If the task has not terminated within 2 seconds, then the subtask will be physically detached. A force option should be considered a last resort option.