Contents

Introduction

Welcome to the Mainframe Cloud (MfC) RunTime Adaptor Db2 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 Db2 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
- DB2 SQL

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.

Writing Apps

To learn how to write web apps for the MfC WS Agent using the MainSpace SDK, refer to the MfC RunTime Adaptor User Guide on the Mainframe Cloud User Resources page.

MainSpace SDK

The MainSpace SDK allows web app developers to code web service calls using MainSpace JavaScript functions instead of URL GET / POST request structures.

This simplifies development in the following ways:

- Defined functions give app developers the self-documenting benefit that comes with IntelliSense® code-completion.

- The MainSpace SDK handles XMLHttpRequest request and response management, Authentication Token control, and Session Identifier management; allowing app developers to concentrate on business logic only.
- The MainSpace SDK includes Db2 functions to group CONNECT, SELECT/UPDATE, COMMIT, and DISCONNECT calls, in one API function call.

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/sql?sql=ssid:dbbg
```

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

POST url:port/logoff
{ “session”:n }

POST url:port/sql
{ “sql”:“ssid:dbbg”, “session”:n }
```

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/sql?sql=ssid:dbbg

POST url:port/sql
{ “sql”:“ssid:dbbg”, “session”:n }
```

The first parameter is the action command.

- For SQL, it is “sql”

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

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, "sql":"ssid:dbbg"

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:

  { "sql":"ssid:dbbg", "session":n } is identical to

  { "session":n, "sql":"ssid:dbbg" }

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 disconnect*** *from Db2* nor will it terminate the web service application task associated with the Db2 session. For example:

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

or

POST url:port/sql
{ “connection”:“close”, “sql”:“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”   ],
   “sqlresp”:{sql-web-service-response}
}
```

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

- SQL requests are specified as: “sql”:“sql_statement”
- SQL response returned is: “sqlresp”:{JSON structure specific to SQL 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 Db2 tables, so it can take advantage of several SQL 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 Db2, 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 Db2.

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 send multiple requests to form a logical session. This may correlate with a Db2 session or it may encompass multiple Db2 sessions. For instance, a logical session (from the web application point of view) could be:

> Db2 connect DB2A, multiple SQL statements, disconnect, connect DB2B, SQL statements, disconnect, logoff

In this example, there are 2 distinct Db2 sessions, but there is one overall session operating under a single task covering both Db2 sessions.

The logoff request will terminate the associated socket connection (that the logoff request was received on), disconnect (the currently connected) Db2 and terminate the application task session. A Db2 disconnect request does not terminate the application task nor the socket session. The web application may want to reconnect to Db2 again or to a different Db2 system, or even to call a different web service altogether. The sequence of web service calls for this example could be:

```
POST url/sql ssid:dbbg
POST url/sql sessid=x, UPDATE ...
POST url/sql sessid=x, COMMIT
POST url/sql sessid=x, DISCONNECT
POST url/sql sessid=x, ssid:dbbc
POST url/sql sessid=x, UPDATE …
POST url/sql sessid=x, COMMIT
POST url/sql sessid=x, DISCONNECT
POST url/logoff sessid=x
```

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.

DB2 Web Service

SQL Calls

An SQL web service call submits a single SQL statement to Db2 on the mainframe. The first SQL statement that must be issued is the Db2 connection statement.

This has the form:

SSID:xxxx where xxxx is the Db2 subsystem identifier

The structure of GET and POST requests for a Db2 connection are:

GET url:port/sql?sql=ssid:dbbg

or

POST url:port/sql
{“sql”=”ssid:dbbg”}

Of course, if an existing logical session is to be used, then add a session=n tag as applicable.

After a successful connection, subsequent web service calls can issue other valid SQL statements. The second last statement issued should be a ‘DISCONNECT’ request. This will terminate the session with Db2. The final request should be a ‘LOGOFF’ request. A logoff request will issue a DISCONNECT Db2 if Db2 is still connected at the time. Change the path to ‘logoff’ and specify the required session identifier. For example:

GET url:port/logoff?session=2

If there are any pending updates, a ‘COMMIT’ SQL statement must be issued to commit the updates permanently. If a COMMIT is not done before the connection is terminated, then any updates will be rolled back to the original setting. A ‘ROLLBACK’ SQL statement can be explicitly issued if required to rollback any updates.

SQL Request JSON Structure

All HTTP POST requests need to supply a JSON request structure, consisting of the SQL statement to be executed.

```
{ “sql”:“sql_statement”, “session”:n }
```

SQL Response JSON Structures

The Db2 SQL web service returns a “sqlresp”:{JSON_structure}. This is inserted into the outer JSON structure described earlier in this document.

Non-Select Statements

```
“sqlresp”:{
    “sql”:“sql_statement”,
    “message”:[
      “message 1”,
      “message n”
    ],
    “sqlcode”:sql_code,
    “sqlstate”:“sql_state”
}
```

The SQL statement received on the request is echoed back on the response. The message array consists of any messages from the web service application. There is also a message array in the outer JSON structure – this contains messages attributed to the general request and not to the individual web service itself. Messages from the individual web service are included in their own message array within their own JSON sub-structure.

Select Statements


```
{ "sqlresp":{
    "sql": "select empno, firstnme from dsn81110.emp where empno <= '000020'",
    "resultset":[{
      "column":[
        { "title":"EMPNO",
          "type":"CHAR",
          "nullable":false,
          "length":6,
          "ccsid":1047
        },
        { "title":"FIRSTNME",
          "type":"VARCHAR",
          "nullable":false,
          "length":12,
          "ccsid":1047
        }
      ],
      "row":[
        {"EMPNO":"000010","FIRSTNME":"CHRISTINE"},
        {"EMPNO":"000020","FIRSTNME":"MICHAEL"   }
      ]
    }],
    "firstrow":1,"lastrow":2,"more":false,
    "sqlcode":0
}}
```


The possible tags in a column construct include title, type, length, nullable, format, precision, scale, ccsid and unsupported.

The current version of the MfC WS Agent does not support Db2 types XML, GRAPHIC, VARGRAPHIC and DBCLOB.

Binary fields are returned in BASE64 format and a “format”:“base64” tag is included. Binary types are BINARY, VARBINARY and BLOB.

Other types which ordinarily should be standard character format, if these contain non-standard EBCDIC printable characters, then these will be returned in either ASCII format (if they are valid ASCII) or in base64 format. A suitable message will also be returned to inform the application of this fact.

In the second last entry in the example above, "firstrow":1,"lastrow":2,"more":false, this relates to blocking controls, not currently implemented in the WS Agent.

The response JSON is not “pretty printed”. Browsers tend to strip white space and line formatting controls from JSON structures. If you wish to read the JSON output in a more readable format, a good suggestion is to use one of the many formatters off the web. Here is one – just copy and paste to - https://jsonformatter.curiousconcept.com/

SQL Response Column Attributes

The "column" section of the JSON output lists the attributes of the returned columns. The returned columns are specified in the Select statement. The following attributes are returned for each column. Attributes without a value are suppressed.

ccsid	Indicates the coded character set identifier of the column.
format	See BASE64 Handling below.
length	Indicates the maximum number of characters for the column.
nullable	Value of TRUE or FALSE, indicates whether a null value is allowed or not.
precision	For numeric columns only, indicates the maximum number of digits for the column, i.e. 1234567.89 has a precision of 9.
scale	For numeric columns only, indicates the maximum number of decimal places for the column, i.e. 1234567.89 has a scale of 2.
title	The name of the column.
type	Indicates the column data type. These are detailed in IBM documentation.
unsupported	Value of TRUE indicates that the Db2 type of this column is unsupported.

BASE64 Handling

The return of non-printable characters in a JSON response structure is not supported. When a Db2 column field contains unprintable characters, the MfC WS Agent will return the value in BASE64 format.

Certain Db2 types (BINARY, VARBINARY and BLOB) are intended to be binary and therefore will likely contain unprintable characters. There are also other Db2 types such as plain character that can also contain unprintable characters. In both of these instances, the Agent will return binary data in BASE64 format.

For binary types, the agent will include a “format”:“base64” tag in the column construct for the appropriate column name and convert the resultant value to BASE64 format.

For character types where the value contains unprintable characters, the value will be converted to BASE64 format and an informative message will be returned.

For example below, this message will be returned indicating that the value of column ZZCHAR in the returned row 1 contains unprintable characters and has been converted to BASE64 format.

```
{ "ZZCHAR": "P1GZgUBAQEA" }

"message":[
  "Field ZZCHAR in row 1 has unprintable characters, converted to base64 format"
]
```

ASCII Handling

If a character column field has non-printable EBCDIC characters but it does contain valid ASCII characters, then the ASCII version of the column field will be returned, and this informative message will be returned:

"Field ZZVARCHAR in row 1 has valid ASCII characters, converted to ASCII"

The CCSID column attribute will typically indicate if a column field is in ASCII format.

Bundled SQL Option

The normal operation is to issue Db2 connect, a number of SQL statements, Db2 disconnect one at a time. Each request is a separate web service request with each receiving its own response.

A bundled SQL option is also available. This entails combining a Db2 connect, a single SQL statement, a commit statement and a Db2 disconnect into a single web service request. The overall benefit for the bundled request is reduced elapsed time and reduced bandwidth used.

The applicable parameters to invoke bundled SQL:

1. sql - existing parameter. SQL statement to be executed.
2. ssid - Db2 SSID to connect to.
3. logoff – a value of 0, 1, 2, 3 or not specified. This indicates whether to logoff the session after a bundled SQL request, depending on the resultant SQLCODE. A value of 0 or not specified are identical and is the default.

A request is considered a bundled request if both of the following are true.

- Both sql and ssid tags are specified in the request JSON. The logoff tag is optional.
- New session or an existing session that is not currently connected to Db2.

A bundled request will perform a Db2 connect, followed by the SQL statement, followed by a SQL commit, followed by a Db2 disconnect. Depending on the resultant SQLCODE and the value of the logoff tag, a final session logoff may or may not be done.

Logoff tag	SQLCODE = 0	SQLCODE > 0	SQLCODE < 0
0	C + D	C + D	N
1	C + D + L	C + D	N
2	C + D + L	C + D + L	N
3	C + D + L	C + D + L	R + D + L

C = Implicit Commit.
R = Implicit Rollback.
D = Disconnect. Commit or rollback will be performed.
L = Logoff. Disconnect automatic. Commit or rollback will be performed.
N = No action. The SQL statement failed, session and connection still intact.

The value of the logoff tag:

- 0 No logoff
- 1 Logoff if SQLCODE = 0
- 2 Logoff if sqlcode is positive, i.e. if SQLCODE >= 0
- 3 Always logoff

For an error condition (SQLCODE is negative), except for option “logoff”=”yes”, the bundled request will be terminated at that point of failure and the Db2 connection (if the connection succeeded) and the session will remain intact. For “logoff”=”yes” option, the session will be logged off completely.

Automatic commit will be enforced for all non-error requests (i.e. SQLCODE >= 0).

For conditions where column contents are converted to ASCII or Base64 due to unprintable characters, this results in SQLCODE=1 returned. For the purposes of logoff of bundled SQL requests, these conditions are treated the same as for SQLCODE=0.

Example request JSON –

{ “sql”: “sql_statement”, “ssid”: “DBBG”, “logoff”: 3 }

The response will be the same as would be returned by executing the SQL statement on its own. If the session is logged off, then no sessionid will be returned.