A web server extension for M/Caché/IRIS and YottaDB
mg_web

M/Gateway Developments Ltd.
Chris Munt

Revision History:
21 July 2020
3 August 2020
17 August 2020
29 August 2020
30 October 2020
6 November 2020
12 November 2020
20 November 2020
29 January 2021
20 April 2021
23 April 2021

Contents

1 Introduction

mg_web provides a high-performance minimalistic interface between three popular web servers (Microsoft IIS, Apache and Nginx) and M-like DB Servers (YottaDB, InterSystems IRIS and Cache).

HTTP requests passed to the DB Server via **mg_web** are processed by a simple function of the form:

```
Response = DBServerFunction(CGI, Content, System)
```

Where *CGI* represents an array of CGI Environment Variables, *Content* represents the request payload and *System* is reserved for **mg_web** use.

A simple 'Hello World' function would look something like the following pseudo-code:

```
DBServerFunction(CGI, Content, System)
{
     // Create HTTP response headers
     Response = "HTTP/1.1 200 OK" + crlf
     Response = Response + "Content-type: text/html" + crlf
     Response = Response + crlf
     //
     // Add the HTML content
     Response = Response + "<html>" + crlf
     Response = Response + "<head><title>" + crlf
     Response = Response + "Hello World" + crlf
     Response = Response + "</title></head>" + crlf
     Response = Response + "<h1>Hello World</h1>" + crlf
     return Response
}
```

In production, the above function would, of course, be crafted in the scripting language provided by the DB Server.

2 Web Server Components

In this section we discuss the process for building and configuring the **mg_web** component for all supported web servers.

2.1 mg_web for Microsoft IIS

mg_web for IIS is implemented as an *IIS Native Module*.

2.1.1 Building from source

It is assumed that you have Visual C++ installed.

Copy the contents of **/src/** and **/src/iis/** to a directory of your choice. You should now have the following files in that directory.

```
Makefile.win
mg_web.c
mg_websocket.c
mg_websocket.h
mg_web.h
mg_websys.h
mg_web_iis.cpp
```

To build **mg_web** for IIS (**mg_web_iis.dll):**

```
nmake -f Makefile.win
```

The following command will simply copy the module (**mg_web_iis.dll**) to the **c:\inetpub\mgweb** directory. You will need to create this directory first.

```
nmake -f Makefile.win install
```

2.1.2 Web Server configuration

The **mg_web** module must be registered as an IIS *'Native Module'*.

Open the **IIS Control Panel** with focus on the root of the IIS installation in the left-hand panel. Open the **Modules** Control Panel.
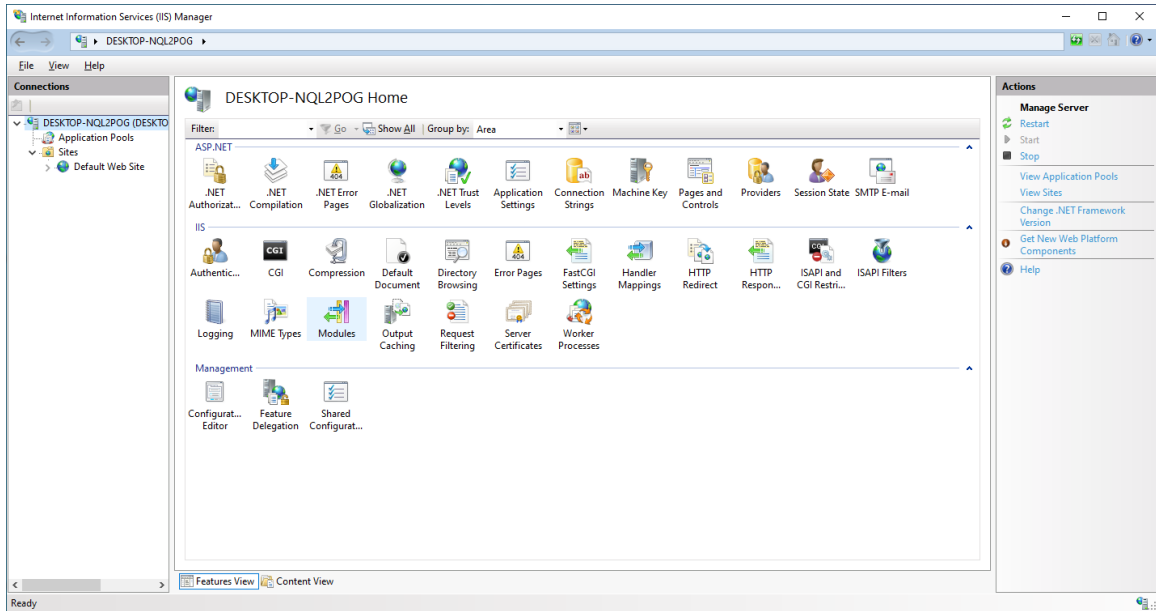
The *Modules* Control Panel. Choose the '*Configure Native Modules*' option in the right-hand panel.
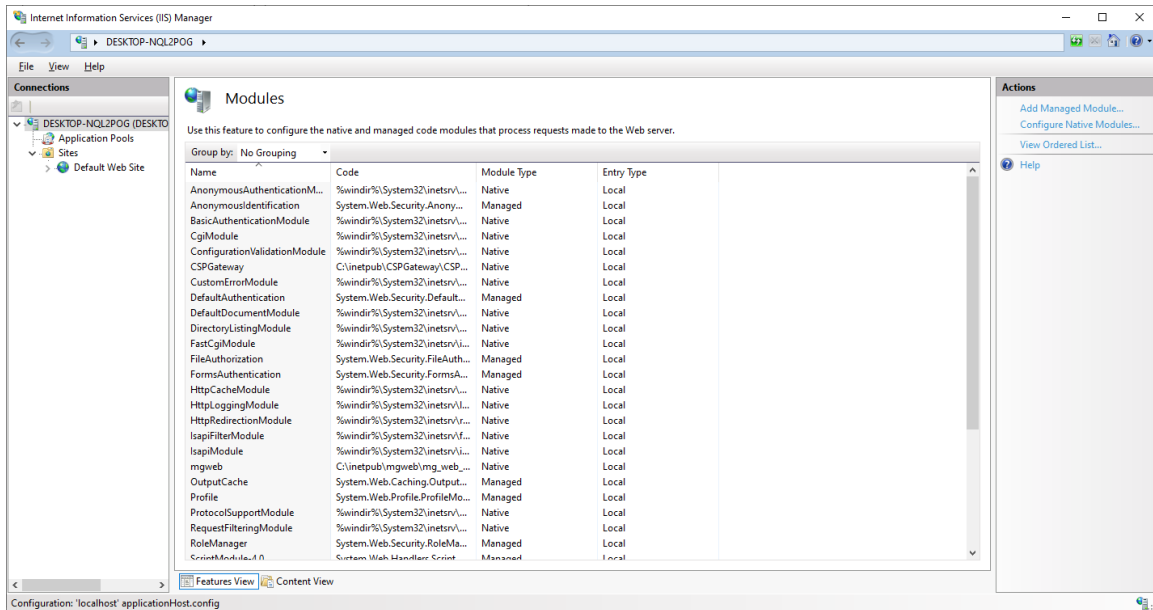
The '*Configure Native Modules*' Control Panel. Note the *Register* button.
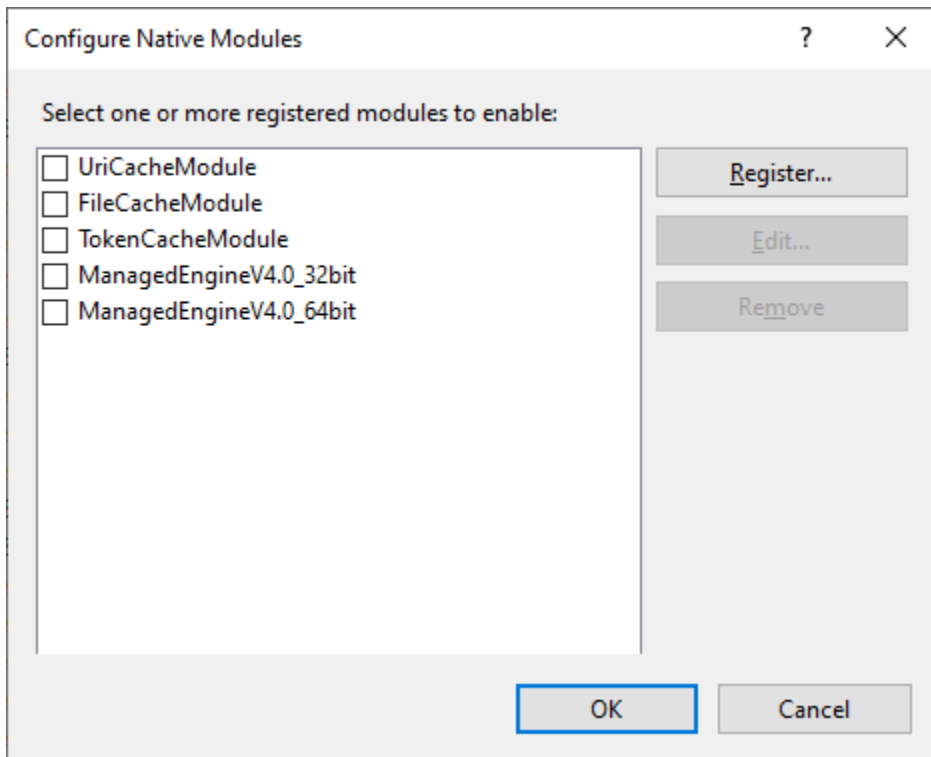
Press the **Register** button to add the **mg_web** module for IIS (**mg_web_iis**).
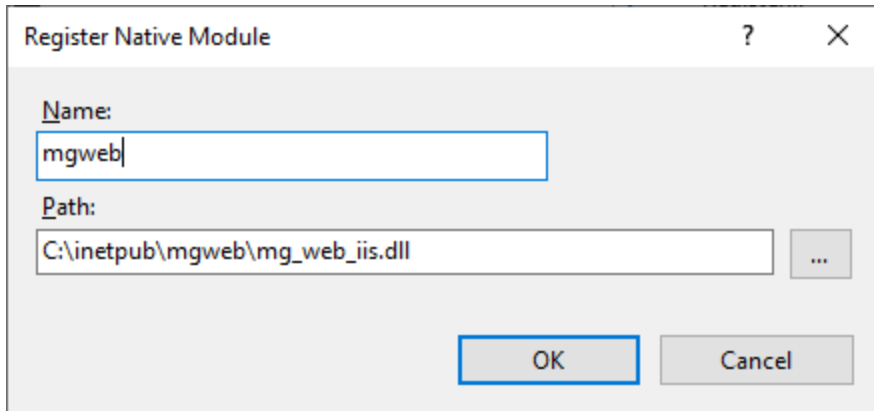
You can assign a name of your choice to the registration – **mgweb** is used in the above example. When the **mg_web** module is registered it can be associated with a particular (virtual) application path and/or specific file types.

To create a new virtual path for applications, right-click on the appropriate web site in the left-hand panel and choose the '**Add Application**' option. In the example below, virtual application path **mgweb** was added beneath the **Default Web Site**.
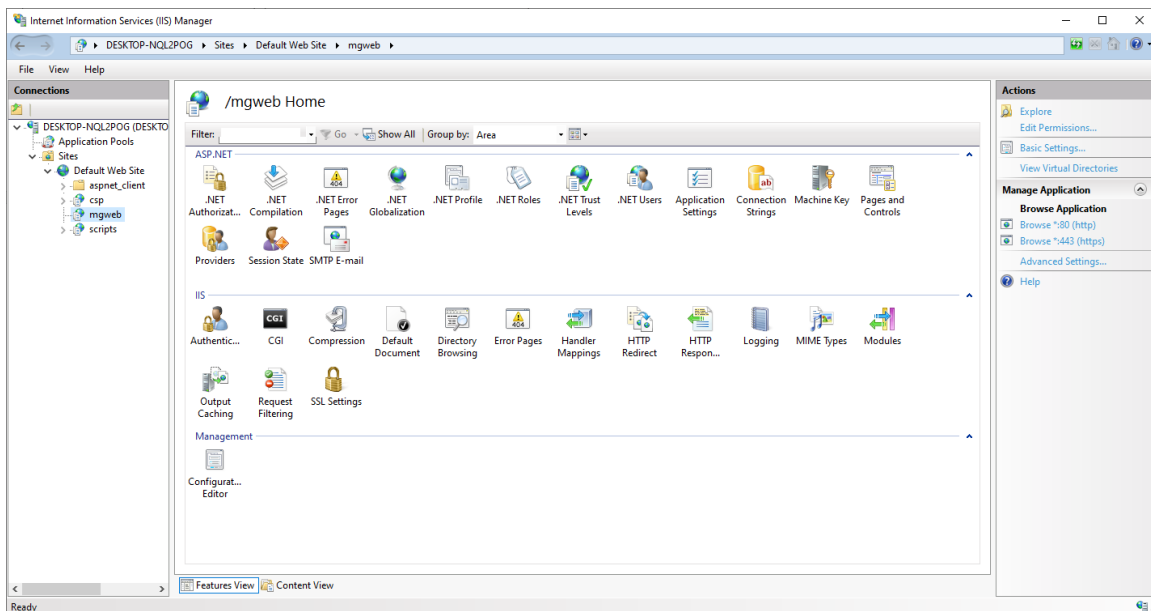
The properties for the virtual application path are shown in the following example. You have to assign an alias (e.g. *mgweb*) and choose an *application pool* to process **mg_web** requests (again named *mgweb* in this example).

Having created a virtual application path for hosting **mg_web** requests, the next task is to map specific file types to the **mg_web** extension. In the following example, we will configure IIS to pass all requests for files of type *.mgw* to **mg_web** for processing.

With the focus on the virtual application path previously created (*mgweb*) in the left-hand panel, open the '*Handler Mappings*' Control Panel and choose '*Add Module Mapping*' in the right-hand panel.
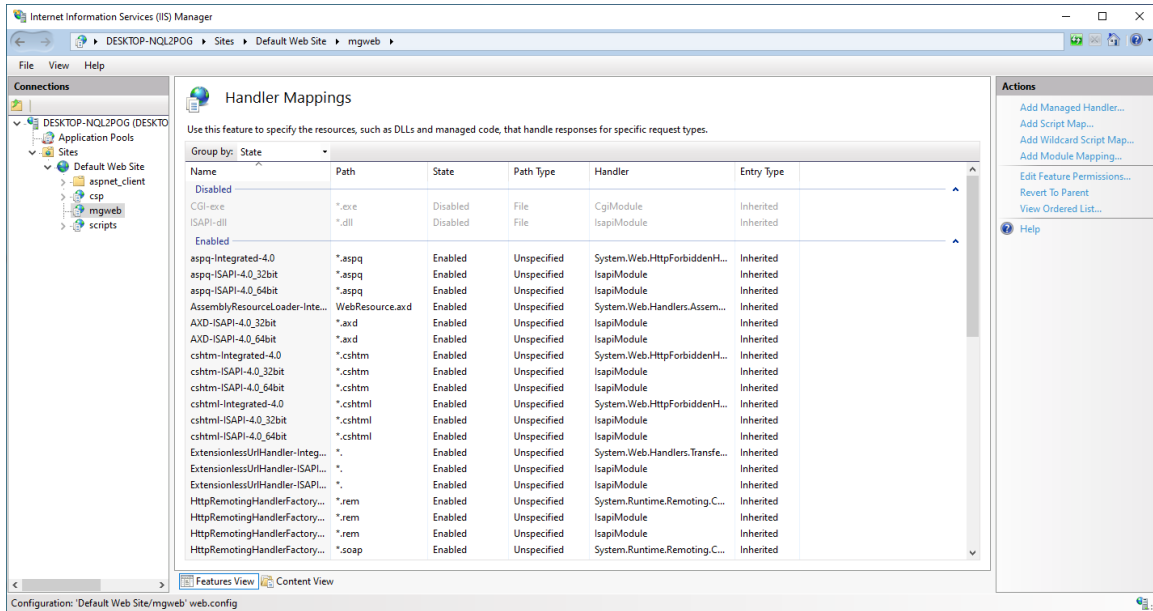
The module name is **mgweb** and we wish to process all files of type *.mgw* with **mg_web**
This is defined in the *Request Path* text box. You can name the Module Mapping with a
name of your choice – **mgweb** is used in the example below.
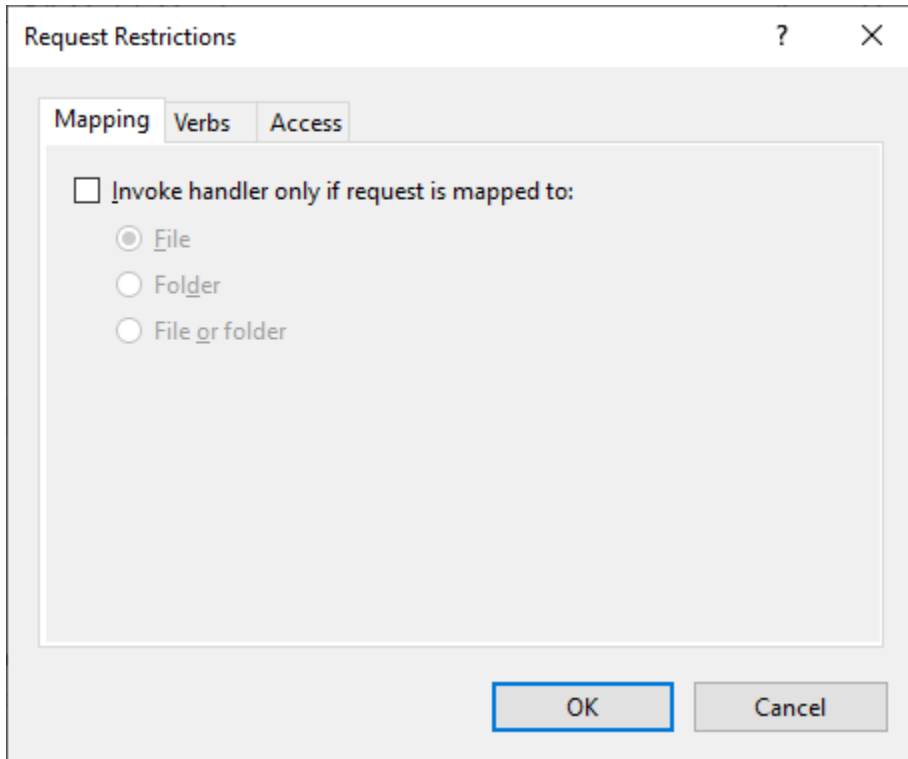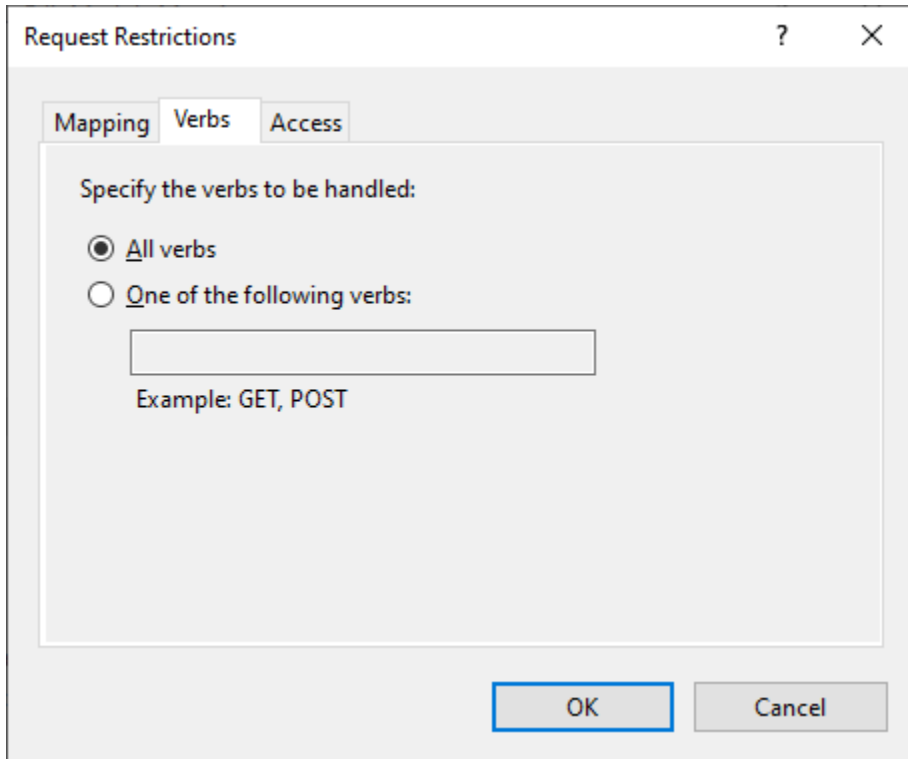
Press the *Request Restrictions* button and make sure that '*Invoke handler only if request is mapped to'* is unchecked. Web resources served my **mg_web** do not physically exist on the web server.

Check that all verbs (i.e. HTTP methods) can be served by **mg_web**.
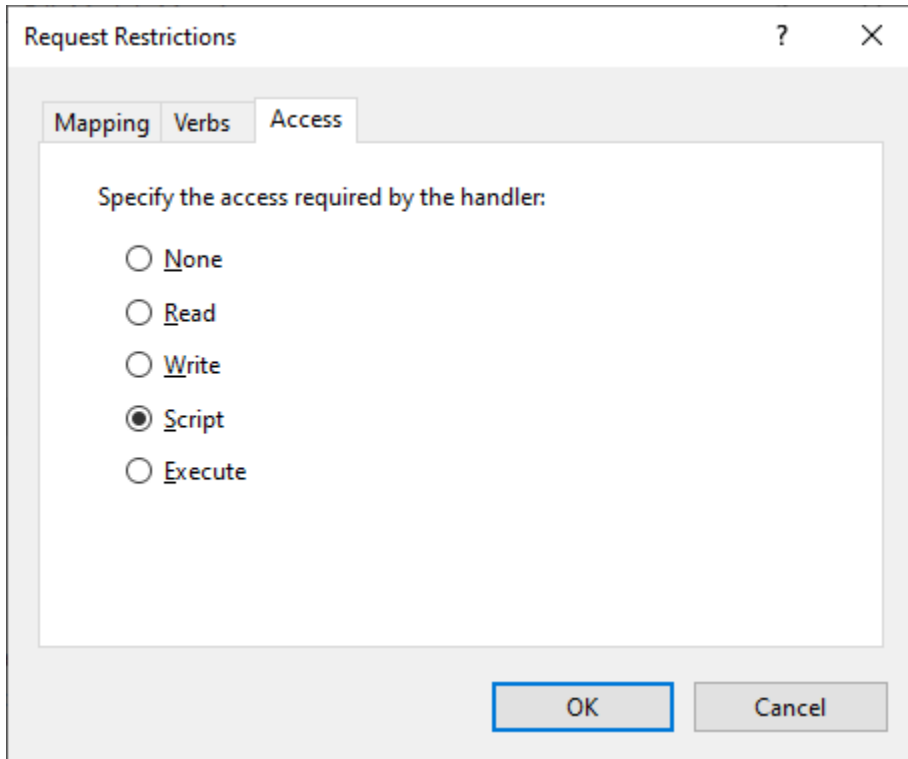
Check that '*Script'* is selected in the *Access* panel.

The **mg_web** configuration and log files (**mgweb.conf** and **mgweb.log**) will be expected to exist in the same directory as that hosting the **mg_web** module. The IIS worker processes must have permission to access these files. Create these files and use the following commands to grant full access to IIS.

```
cacls C:\inetpub\mgweb\mgweb.conf /E /G IIS_IUSRS:F
cacls C:\inetpub\mgweb\mgweb.log /E /G IIS_IUSRS:F
```

Final contents of the **mg_web** directory (**C:\inetpub\mgweb**):

```
mg_web_iis.dll
mgweb.conf
mgweb.log
```

Finally, restart IIS from the main Windows Services Control Panel (***World Wide Web Publishing Service***).
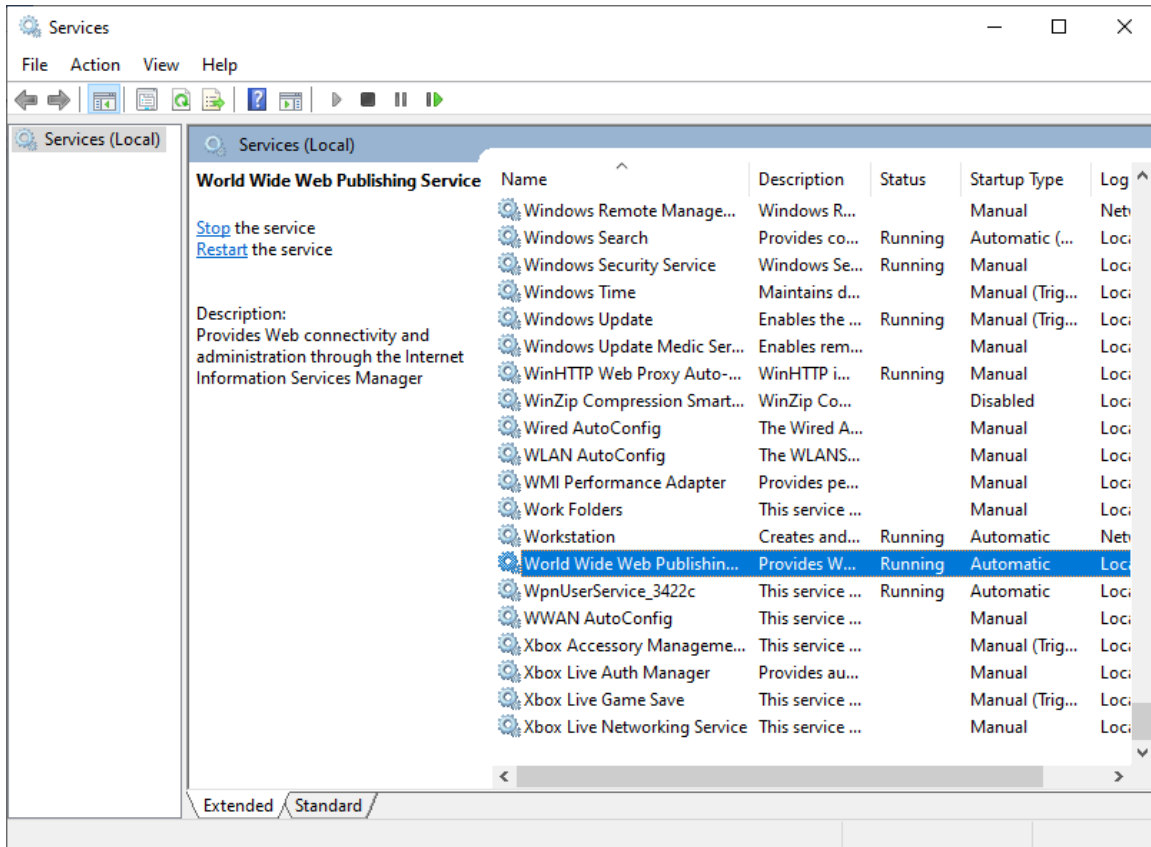
IIS will now pass all requests for files of type *.mgw* in path *mgweb* to **mg_web** for processing. For example:

http://localhost/mgweb/myfile.mgw

2.2 mg_web for Apache

mg_web for Apache is implemented as an *Apache Extension Module*.

2.2.1 Building from source

It is assumed that you have a suitable C/C++ compiler installed. For example, GCC for Linux and Visual C++ for Windows.

Copy the contents of **/src/** and **/src/apache/** to a directory of your choice. You should now have the following files in that directory.

```
mg_web.c
mg_websocket.c
mg_websocket.h
mg_web.h
mg_websys.h
mg_web_apache.c
```

To build **mg_web** for Apache (**mg_web_apache.so** or **mg_web_apache.dll**)**:**

Using the Apache extension compiler tool - *APache eXtenSion tool* (**apxs**):

```
apxs -a -i -c mg_web_apache.c mg_web.c mg_websocket.c
```

This will compile the **mg_web** extension for Apache, install the module in the Apache *modules* directory, and add the following line to the Apache configuration file (***httpd.conf***):

```
LoadModule mg_web_module     modules/mg_web_apache.so
```


2.2.2 Web Server configuration

Check that the **mg_web** module is registered in the Apache configuration file (***httpd.conf***).

```
LoadModule mg_web_module     modules/mg_web_apache.so
```


Add the full path of the **mg_web** configuration file (*mgweb.conf*) and log file (*mgweb.log*) to the Apache configuration file (***httpd.conf***). For example:

```
MGWEBConfigFile c:/Apache2433/conf/mgweb.conf
MGWEBLogFile c:/Apache2433/logs/mgweb.log
```

Create a location through which **mg_web** requests will be processed. For example, **mg_web** can be active for a whole path (*mgweb* in this example):

```
<Location /mgweb>
    MGWEB On
</Location>
```

Alternatively, **mg_web** can be set to configure only specific file types.

```
<Location /mgweb>
    MGWEBFileTypes .mgw .mgweb
</Location>
```

Finally, restart Apache with the new configuration and the web server will now (for example) pass all requests for files of type *.mgw* in path *mgweb* to **mg_web** for processing.

http://apachehost/mgweb/myfile.mgw

2.3 mg_web for Nginx

mg_web for Nginx is implemented as a *Nginx Addon Module*. Unlike the other web server solutions where **mg_web** is created as a dynamically loaded library, **mg_web** functionality is built directly into the Nginx core executable.

2.3.1 Building from source

It is assumed that you have a suitable C/C++ compiler installed. For example, GCC for Linux and Visual C++ for Windows. Additionally, on Windows Nginx is built using the MSYS toolkit and that should be installed. The Nginx instructions for building this web server under Windows can be found here.

http://nginx.org/en/docs/howto_build_on_win32.html

Copy the contents of **/src/** and **/src/nginx/** to a directory of your choice For example, /**opt/mgweb/**. You should now have the following files in that directory.

```
config
mg_web.c
mg_websocket.c
mg_websocket.h
mg_web.h
mg_websys.h
mg_web_nginx.c
```

To build **mg_web** into Nginx:

For UNIX systems, add the **mg_web** module directory to the pre-build configuration step. For example:

```
./configure --prefix=/opt/nginx1180 \
            --with-threads \
            --add-module=/opt/mgweb
```

Note that the '--with-threads' option must be included if **mg_web** is to take advantage of Nginx thread pooling (recommended).

Having run the configuration step, the Nginx web server with **mg_web** included can be built using:

```
make
make install
```

For Windows systems, using the MSYS environment, the process is very similar. Add the **mg_web** module directory to the pre-build configuration step. For example:

```
auto/configure --with-cc=cl --builddir=objs --prefix= \
--conf-path=conf/nginx.conf --pid-path=logs/nginx.pid \
--http-log-path=logs/access.log                    --error-log-
path=logs/error.log \
--sbin-path=nginx.exe \
--http-client-body-temp-path=temp/client_body_temp \
--http-proxy-temp-path=temp/proxy_temp \
--http-fastcgi-temp-path=temp/fastcgi_temp \
--with-cc-opt=-DFD_SETSIZE=1024 \
--without-http_rewrite_module \
--without-http_gzip_module \
--with-select_module \
--add-module=/opt/mgweb
```

Having run the configuration step, the Nginx web server with **mg_web** can be built using:

```
nmake -f objs/Makefile
```

2.3.2 Web Server configuration

Add the full path of the **mg_web** configuration file (**mgweb.conf**) and log file (**mgweb.log**) to the Nginx configuration file (*nginx.conf*). These directives should be added to the *http* section of *nginx.conf*. For example:

```
MGWEBConfigFile /opt/nginx1180/conf/mgweb.conf
MGWEBLogFile /opt/nginx1180/logs/mgweb.log
```

Create a location through which **mg_web** requests will be processed. These directives should be added to the *server* section of *nginx.conf*. For example, **mg_web** can be active for a whole path:

```
        location /mgweb {
            MGWEB On;
            MGWEBThreadPool default;
        }
```

Alternatively, **mg_web** can be set to configure only specific file types.

```
        location /mgweb {
            MGWEBFileTypes .mgw .mgweb;
            MGWEBThreadPool default;
```

```
        }
```

Note that in both cases **mg_web** is configured to use a Nginx thread pool called *default*.

Finally, restart Nginx with the new configuration and the web server will now (for example) pass all requests for files of type *.mgw* in path *mgweb* to **mg_web** for processing.

http://nginxhost/mgweb/myfile.mgw

3 Installing the DB Superserver

The DB Server components are held in the M/Gateway *Service Integration Gateway* (**mgsi**) GitHub repository. These components are two DB Server routines (i.e. M routines): **%zmgsi** and **%zmgsis**. In this section we will look at the procedure for installing them.

https://github.com/chrisemunt/mgsi

3.1 Installation for InterSystems Caché or IRIS

Log in to the Manager Namespace (%SYS) and install the **zmgsi** routines held in the **/isc/zmgsi_isc.ro** package (these are found in the **mgsi** GitHub repository).

```
do $system.OBJ.Load("/isc/zmgsi_isc.ro","ck")
```

Change to your development UCI and check the installation:

```
do ^%zmgsi

M/Gateway Developments Ltd - Service Integration Gateway
Version: 4.2; Revision 20 (20 April 2021)
```

3.2 Installation for YottaDB

Install the two **zmgsi** routines held in the **/yottadb/** directory of the **mgsi** GitHub repository.

The instructions given here assume a standard 'out of the box' installation of **YottaDB** deployed in the following location:

```
/usr/local/lib/yottadb/r130
```

The primary default location for routines:

```
/root/.yottadb/r1.30_x86_64/r
```

Copy all the routines (i.e. all files with an 'm' extension) held in the GitHub **/yottadb** directory to:

```
/root/.yottadb/r1.30_x86_64/r
```

Change directory to the following location and start a **YottaDB** command shell:

```
cd /usr/local/lib/yottadb/r130
./ydb
```

Check the installation:

```
do ^%zmgsi

M/Gateway Developments Ltd - Service Integration Gateway
Version: 4.2; Revision 20 (20 April 2021)
```

Note that the version of **zmgsi** is successfully displayed.

3.3 Starting the DB Superserver

The default TCP server port on which the DB Superserver (**%zmgsi)** listens is **7041**. If you wish to use an alternative port then modify the following instructions accordingly.

3.3.1 Starting the DB Superserver from the DB command prompt

- For InterSystems DB servers the concurrent TCP service should be started in the **%SYS** Namespace.

Start the DB Superserver using the following command:

```
do start^%zmgsi(0)
```

To use a server TCP port other than 7041, specify it in the start-up command (as opposed to using zero to indicate the default port of 7041).

3.3.2 Starting YottaDB Superserver processes via the xinetd daemon

Network connectivity to **YottaDB** is managed via the **xinetd** service. First create the following launch script (called **zmgsi_ydb** here):

```
/usr/local/lib/yottadb/r130/zmgsi_ydb
```

Content:

```
#!/bin/bash
cd /usr/local/lib/yottadb/r130
export ydb_dir=/root/.yottadb
export ydb_dist=/usr/local/lib/yottadb/r130
export
ydb_routines="/root/.yottadb/r1.30_x86_64/o*(/root/.yottadb/r1.30_x86_6
4/r /root/.yottadb/r) /usr/local/lib/yottadb/r130/libyottadbutil.so"
```

```
export ydb_gbldir="/root/.yottadb/r1.30_x86_64/g/yottadb.gld"
$ydb_dist/ydb -r xinetd^%zmgsis
```

Create the **xinetd** script (called **zmgsi_xinetd** here):

```
/etc/xinetd.d/zmgsi_xinetd
```

Content:

```
service zmgsi_xinetd
{
    disable         = no
    type            = UNLISTED
    port            = 7041
    socket_type     = stream
    wait            = no
    user            = root
    server          = /usr/local/lib/yottadb/r130/zmgsi_ydb
}
```

- Note: sample copies of **zmgsi_xinetd** and **zmgsi_ydb** are included in the **/unix** directory of the **mgsi** GitHub repository.

Edit the services file:

```
/etc/services
```

Add the following line to this file:

```
zmgsi_xinetd            7041/tcp                        # ZMGSI
```

Finally restart the **xinetd** service:

```
/etc/init.d/xinetd restart
```

4 General mg_web configuration (mgweb.conf)

The **mg_web** configuration file (*mgweb.conf*) contains the instructions for connecting to each DB Server and which web paths should be routed to each DB Server.

The following configuration parameters are specified at the global level and apply to the whole **mg_web** installation.

Timeouts

```
timeout 30
```

This is the amount of time (in seconds) that **mg_web** will wait for a response to be returned by the DB Server.

The Event Log

You can control the amount of information written to the Event Log:

```
log_level <directives>
```

Where the *directives* may include:

- **e**   Log error conditions.
  f   Log the basic framing information for the request and response buffers.
- **t**   Log the contents of the request and response data buffers transmitted between **mg_web** and the DB Server.
- **w**   Log the final response buffers dispatched from **mg_web** to the Web Server, showing the chunked transfer protocol where applicable.

Example:

```
log_level eftw
```

CGI Environment Variables

You can also define lists of CGI environment variables to be sent to the DB Server with each request. For example, the following directive will instruct **mg_web** to send to the DB Server all CGI environment variables derived from client HTTP request headers (***HTTP****) and the web server ***Server Software*** with each request.

```
<cgi>
   HTTP*
```

```
    SERVER_SOFTWARE
</cgi>
```

Note that, by default, **mg_web** will only send the following CGI environment variables to the DB Server with each request.

```
REQUEST_METHOD
SCRIPT_NAME
QUERY_STRING
```

Chunking

It is also possible to control the level at which HTTP chunked transfer of the response payload will occur. By default, **mg_web** will read approximately 64K of response data from the DB Server before switching on chunking for the response. Up to that point the whole response will be cached in a single buffer and a 'Content-Length' header added. The use of chunked transfer can be completely disabled as follows:

```
chunking off
```

Alternatively, a size threshold can be defined after which chunking will take place. The value for this field can be specified in Bytes, Kilobytes (KB) or Megabytes (MB). For example, to instruct **mg_web** to use chunked transfer for response payloads exceeding 250 Kilobytes:

```
chunking 250KB
```

- Note that if the application on the DB Server returns a 'Content-Length' field, **mg_web** will use it. Take care to ensure that the value supplied is accurate!

Working Buffer Size

By default, **mg_web** will allocate enough internal working buffer space to accommodate the request payload or 128KB, whichever is the greater. This same buffer is used to receive the response payload from the DB Server. While **mg_web** can receive response payloads that exceed the size of the pre-allocated working buffer, the following configuration parameter can be used to set a minimum size for the buffer.

```
request_buffer_size 500KB
```

The above parameter, when added to the global section of the configuration file, will instruct **mg_web** to allocate at least 500KB of working buffer space. You would typically increase the size of this buffer to optimize installations that routinely return more than 128KB of response payload (including HTTP response headers). The effects

---

of such optimizations are particularly significant for configurations that communicate with local DB Servers via their API.

Custom Error Pages

Finally, in the global section of the configuration, it is possible to define custom HTML pages to be returned when errors are encountered by **mg_web**. There are three types of error condition for which it is possible to define a custom response.

DB Server is currently unavailable (i.e. a connectivity problem).
Example:

```
custompage_dbserver_unavailable http://webserver/dbserver_unavailable.html
```

DB Server is currently busy (i.e. capacity of installation exceeded).
Example:

```
custompage_dbserver_busy http://webserver/dbserver_busy.html
```

DB Server is currently disabled (i.e. DB Server is deliberately taken out of service).
Example:

```
custompage_dbserver_disabled http:// webserver/dbserver_disabled.html
```

4.1 Defining Servers

The following examples will illustrate how DB Server access should be defined for **mg_web**.

Network based access to InterSystems IRIS (or Cache) listening on TCP port 7041:

```
<server local>
   type IRIS
   host localhost
   tcp_port 7041
   username _SYSTEM
   password SYS
   namespace USER
</server>
```

API based access to InterSystems IRIS (or Cache):

```
<server local>
   type Cache
   path /opt/cache20181/mgr
   username _SYSTEM
   password SYS
   namespace USER
</server>
```

Network based access to YottaDB listening on TCP port 7041:

```
<server local>
   type YottaDB
   host localhost
   tcp_port 7041
</server>
```

API based access to YottaDB:

```
<server local>
    type YottaDB
    path /usr/local/lib/yottadb/r130
    <env>
        ydb_dir=/root/.yottadb
        ydb_rel=r1.30_x86_64
        ydb_gbldir=/opt/webapps/yottadb.gld
        ydb_routines=/opt/webapps
        ydb_ci=/usr/local/lib/yottadb/r130/zmgsi.ci
    </env>
</server>
```

The routine interface file (named *zmgsi.ci* in the above example) must contain the following line:

```
ifc_zmgsis: ydb_string_t * ifc^%zmgsis(I:ydb_string_t *,
I:ydb_string_t *, I:ydb_string_t *)
```

4.2 Defining Paths

The following examples will illustrate how web paths should be defined for **mg_web**.

The root path (effectively the default mapping):

```
<location />
    function web^%zmgweb
    servers local
</location>
```

Further examples:

```
<location /mgweb/path1>
    function web1^%zmgweb
    servers local1
</location>

<location /mgweb/path2>
    function web2^%zmgweb
    servers local2
</location>
```

A hierarchical system of inheritance for the paths is applied. For Example:

http://webserver/mgweb/path1/file.mgw

This request will be routed to DB Server **local1**

http://webserver/mgweb/path2/file.mgw

This request will be routed to DB Server **local2**

http://webserver/mgweb/path2/abc/file.mgw

This request will be routed to DB Server **local2**

http://webserver/mgweb/file.mgw

This request will be routed to DB server **local**

---

http://webserver/xyz/file.mgw

This request will be routed to DB server **local** (assuming the web server is configured to pass requests with a path of **/xyz** to **mg_web**).

We now look at the web path configuration parameters in detail.

function

The DB Server function to be invoked for this path. For example:

```
function web^%zmgweb
```

The form of this function is described in more detail in a later section. In brief, web functions should be constructed as follows.

```
web^%zmgweb(%cgi,%content,%system)
    ; process request and generate response
    Quit response
```

servers

One or more DB Server can be defined for the path. For example:

```
server dbserver0 dbserver1 dbserver2 etc …
```

4.2.1 Load balancing and Fail-over

We have already seen that multiple servers can be defined for each path. Additional DB Servers can be used only for the purposes of fail-over (the default) or used for load balancing and fail-over.

```
load_balancing <on|off> (default is off)
```

For example, to enable load balancing amongst the available servers.

```
load_balancing on
```

With load balancing comes the issue of *server affinity*, often known as *sticky sessions*. In other words, once a user has connected to a web application via one particular DB Server, you want them to continue to use that server for the duration of their application session. This can be achieved either via a form/URL variable or via a 'server affinity' cookie.

Server affinity using a variable

```
server_affinity variable:<variable(s)>
```

For example, using a form URL variable called **MyServer**:

```
server_affinity variable:MyServer
```

http://webserver/mgweb/path1/file.mgw?MyServer=dbserver2

DB Servers are numbered from zero (left to right), so the following equivalent URL could be used:

http://webserver/mgweb/path1/file.mgw?MyServer=2

The *server name* and *server number* used is presented to the application code in the **%system** array. For example:

```
%system("server")="dbserver2"
%system("server_no")=2
```

It is possible to specify that multiple variables be used for server affinity. For example:

```
server_affinity variable:MyServer1,MyServer2
```

In this case the DB Server name (or number) can be specified either in form/URL variable *MyServer1* or *MyServer2*.

Server affinity using a cookie

```
server_affinity cookie:<name>
```

For example, using a server affinity cookie called **mgweb_dberver**:

```
server_affinity cookie:mgweb_dbserver
```

If a server affinity cookie is used, **mg_web** will assume responsibility for inserting the appropriate cookie value in the HTTP response headers.

It is possible to specify that both form/URL variables and a cookie be used for server affinity. Examples:

```
server_affinity variable:MyServer cookie:mgweb_dbserver
```

In this case **mg_web** will first look for the specified form/URL variable and then look for the cookie if not found.

```
server_affinity cookie:mgweb_dbserver variable:MyServer
```

In this case **mg_web** will first look for the specified cookie and then look for the form/URL variable if not found.

Finally, load balancing is implemented as simple 'round robin' and in cases where a DB Server becomes unresponsive, the request is failed over to the next DB Server in the list.

4.3 Complete example mgweb.conf

```
timeout 30

<cgi>
   HTTP*
   SERVER_SOFTWARE
</cgi>

<server local>
   type IRIS
   host localhost
   tcp_port 7041
   username _SYSTEM
   password SYS
   namespace USER
</server>

<location />
   function web^%zmgweb
   servers local
</location>
```

4.4 Reporting configuration errors

It is essential that the **mg_web** event log file is specified correctly and that the web server worker processes are granted full access to it as any configuration errors will be reported in the log.

When a web server worker process successfully links to the **mg_web** library, a message such as that shown below will be written to the **mg_web** event log.

```
>>> Time: Thu Jul 23 16:21:44 2020; Build: 1.0.1 pid=9364;tid=27368;
   mg_web: worker initialization
   configuration: C:/inetpub/mgweb/mgweb.conf
```

If a configuration error is detected it will be reported after the initialization message. For example, if parameter *tcp_port* is not specified correctly, a sequence of messages similar to those shown below will be reported.

```
>>> Time: Thu Jul 23 16:21:44 2020; Build: 1.0.1 pid=9364;tid=27368;
    mg_web: worker initialization
    configuration: C:/inetpub/mgweb/mgweb.conf
>>> Time: Thu Jul 23 16:21:44 2020; Build: 1.0.1 pid=9364;tid=27368;
    mg_web: configuration error
    Invalid 'server' parameter 'tcpport' on line 11
```

5 The mg_web DB Server function

The signature of DB Server functions for **mg_web** is as follows:

```
web^%zmgweb(%cgi,%content,%system)
   ; process request and generate response
   Quit response
```

Where:

```
%cgi:     List of CGI Environment Variables
%content: The request payload (if any)
%system:  Read-only system array reserved for mg_web use.
```

Of course, the function may be named as you wish but must match the corresponding *function* entry in the **mg_web** configuration file (*mgweb.conf*).

The *System* array (**%system**) contains the name of the server and path relevant to the current request, as defined in the **mg_web** configuration file (**mgweb.conf**). For example:

```
%system("server")="local"
%system("path")="/"
```

- **Note: It is important that the read-only %*system* array is protected in your application code. It is good practice to extract (from %*system*) any information required by the application then protect it with a DB Server *new* statement. All the examples given here, follow this practice.**

5.1 Helper functions

There are a number of 'helper' functions available to **mg_web** applications. These are described in this section.

5.1.1 Parse application/x-www-form-urlencoded

Parse content of type 'application/x-www-form-urlencoded' OR a QUERY_STRING to return an array of name/value pairs:

```
Set %status=$$nvpair^%zmgsis(.%nv,%content)
```

Where:

```
%nv:      An array of name/value pairs
%content: The request payload or a QUERY_STRING
```

This function will un-escape all components before placing them in the name/value pair array.

5.1.2 Parse multipart/form-data

Parse content of type 'multipart/form-data' to return an array of name/section pairs:

```
Set %status=$$multipart^%zmgsis(.%nv,.%nvhead,%content,%boundary)
```

Where:

```
%nv:       An array of name/section pairs
%nvhead:   An array of section headers
%content:  The multipart request payload
%boundary: The multipart boundary marker
```

This function will generate a name/section pair array from the multipart payload. Under each section name, the section header directives will be placed in the 'headers' array as follows.

```
%nvhead(name,header_directive)=header_value
```

For example, on uploading a simple text file, you would expect to see something like:

```
%nv("MyFile")="Contents of my text file"
%nvhead("MyFile","Content-Disposition")=
                "form-data; name="MyFile"; filename="c:\myfile.txt""
%nvhead("MyFile","Content-Type")="text/plain"
```

5.1.3 Generic content parser

Parse content of type 'application/x-www-form-urlencoded' OR 'multipart/form-data' OR a QUERY_STRING to return an array of name/value (or section) pairs:

```
Set %status=$$content^%zmgsis(.%nv,.%nvhead,%content,.%cgi)
```

Where:

```
%nv:      An array of name/value (or section) pairs
%nvhead:  An array of section headers (where applicable)
%content: The request payload or a QUERY_STRING
%cgi:     List of CGI Environment Variables
```

This function will call **nvpair^%zmgsis()** or **multipart^%zmgsis()** depending on the payload content type.

5.1.4 Decoding URLs

URL decoding function (URL un-escaping):

```
Set %decoded=$$urld^%zmgsis(%encoded)
```

Where:

```
%encoded: URL-escaped item.
%decoded: URL-unescaped item.
```

URL encoding function (URL escaping):

```
Set %decoded=$$urle^%zmgsis(%decoded)
```

Where:

```
%decoded: URL-unescaped item.
%encoded: URL-escaped item.
```

5.1.5 Maximum DB Server string length

Determine the maximum string length for this DB Server installation:

```
Set %max=$$getmsl^%zmgsis()
```

Where:

```
%max:     Maximum string length.
```

5.2 Streaming the response using 'Write' statements (Block mode)

In addition to the scheme whereby the response is returned from the DB Server function as a single string, **mg_web** provides for the response to be streamed back to the client using *Write* statements. Response content can be written out using the *write* procedure provided by **mg_web** or, for InterSystems DB Servers, the *Write* command provided by the embedded DB scripting language.

To indicate that you wish to return the response content as a stream, the DB Server function must first call the *stream* function:

```
Set %str=$$stream^%zmgsis(.%system)
```

Where:

```
%str:        The stream.
%system:     The mg_web reserved system array (passed by reference).
```

The stream variable (i.e. **%str**) must subsequently be returned as the output from your DB Server function.

The **mg_web** *write* procedure is defined as follows:

```
Do write^%zmgsis(.%str,content)
```

Where:

```
%str:        The stream (passed by reference).
content:     The response content to write out.
```

The following two examples will illustrate the two approaches for streaming response content back to the client.

Using the mg_web *write* procedure (InterSystems DB Servers and YottaDB):

```
web^%zmgweb(%cgi,%content,%system)
   set %str=$$stream^%zmgsis(.%system)
   new %system
   do write^%zmgsis(.%str,"HTTP/1.1 200 OK"_$c(13,10))
   do write^%zmgsis(.%str,"Content-type: text/html"_$c(13,10,13,10))
   do write^%zmgsis(.%str,"<html>")
   do write^%zmgsis(.%str,"<head><title>Hello</title></head>")
   do write^%zmgsis(.%str,"<h1>Hello World</h1>")
   do write^%zmgsis(.%str,"</html>")
   quit %str
```

Using the embedded DB Server Write command (InterSystems DB Servers only):

```
web^%zmgweb(%cgi,%content,%system)
   set %str=$$stream^%zmgsis(.%system)
   new %system
   write "HTTP/1.1 200 OK"_$c(13,10)
   write "Content-type: text/html"_$c(13,10,13,10)
   write "<html>"
   write "<head><title>Hello</title></head>"
   write "<h1>Hello World</h1>"
   write "</html>"
   quit %str
```

5.3 Streaming the response using 'Write' statements (ASCII mode)

In addition to the *Block mode* scheme for returning content described in the previous section, **mg_web** provides for ASCII content to be streamed back to the client using DB Server Write commands. The benefit of this mode is that DB Server Write statements can be used for both InterSystems DB Servers and YottaDB. The *Block mode* should always be used when returning binary data.

To indicate that you wish to return the response content as a stream of ASCII characters, the DB Server function must first call the *streamascii* function:

```
Set %str=$$streamascii^%zmgsis(.%system)
```

Where:

```
%str:      The stream.
%system:   The mg_web reserved system array (passed by reference).
```

Response content is then streamed back to the client using DB Server Write statements as shown below.

```
web^%zmgweb(%cgi,%content,%system)
   set %str=$$streamascii^%zmgsis(.%system)
   new %system
   write "HTTP/1.1 200 OK"_$c(13,10)
   write "Content-type: text/html"_$c(13,10,13,10)
   write "<html>"
   write "<head><title>Hello</title></head>"
   write "<h1>Hello World</h1>"
   write "</html>"
   quit %str
```

6 Using WebSockets

WebSockets are supported by **mg_web** according to the following scheme.

The file name in the WebSocket request must indicate the name of the DB Server function implementing the server side of the WebSocket connection. For example:

```
/mgweb/MyWebSocket^MyWebFunctions.mgw
```

This indicates a WebSocket function of: `MyWebSocket^MyWebFunctions`

The signature of this function is the same as for other **mg_web** web functions.

```
MyWebSocket(%cgi,%content,%system)
    set %status=$$websocket^%zmgsis(.%system,0,"")
    new %system
    ; Read from and write to WebSocket client
    quit %status
```

Note that the WebSocket function must first call the websocket initialization function:

```
Set %status=$$websocket^%zmgsis(.%system,binary,options)
```

Where:

```
%status:    The status (should be "").
%system:    The mg_web reserved system array (passed by reference).
binary:     The binary flag (1 or 0).
            Set to 1 to return binary content to the WebSocket client.
options:    Reserved for future use.
```

6.1 A simple WebSocket example

In this simple WebSocket implementation, the DB Server simply echoes back the data sent by the client with the DB Server internal date and time appended. It will return the DB Server version if $zv is entered, and exit if 'exit' is entered.

Web client-side code:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <title>WebSocket Echo</title>
    <script type="text/javascript">
    <!--
      var ws;

      if ((typeof(WebSocket) == 'undefined') &&
          (typeof(MozWebSocket) != 'undefined')) {
        WebSocket = MozWebSocket;
      }

      function init() {
        ws = new WebSocket(((window.location.protocol == "https:") ? "wss:" : "ws:") +
"//" + window.location.host + "/mgweb/MyWebSocket^MyWebFunctions.mgw");
        ws.onopen = function(event) {
          document.getElementById("main").style.visibility = "visible";
          document.getElementById("connected").innerHTML = "Connected to WebSocket
server";
        };
        ws.onmessage = function(event) {
          document.getElementById("output").innerHTML = event.data;
        };
        ws.onerror = function(event) { alert("Received error"); };
        ws.onclose = function(event) {
          ws = null;
          document.getElementById("main").style.visibility = "hidden";
          document.getElementById("connected").innerHTML = "Connection Closed";
        }
      }

      function send(message) {
        if (ws) {
          ws.send(message);
        }
      }
    }
    // -->
    </script>
  </head>
  <body onload="init();">
    <h1>WebSocket Echo</h1>
    <div id="connected">Not Connected</div>
    <div id="main" style="visibility:hidden">
    Enter Message: <input type="text" name="message" value="" size="80"
onchange="send(this.value)"/><br/>
    Server says... <div id="output"></div>
    </div>
  </body>
</html>
```

DB Server-side code:

The function first waits for a single character to arrive with a timeout of 30 seconds. If no data is received within the 30 second period, the function will write a message to the client and wait again. If client data does arrive then it reads the rest of the input (using a 0 second timeout). If the client sends '$zv' then the function returns the version of the DB Server, otherwise it appends the internal date and time ($H) to the data and sends it back to the client. If the client sends 'exit' then the server closes the WebSocket.

```
MyWebSocket(%cgi,%content,%system)
     set %status=$$websocket^%zmgsis(.%system,0,"")
     new %system
loop ; loop
     read *chr:30 if '$test write "Timeout at "_$H do flush^%zmgsis goto loop
     read data:0 set data=$char(chr)_data
     if data="exit" quit %status
     if data="$zv" s data=$zv
     write "DB server response at "_$H_": "_data do flush^%zmgsis
     goto loop
```

WebSocket server processes are never reused by **mg_web**. The process halts when the WebSocket session is complete. Applications can modify the device characteristics of their primary device to suit the needs of the functionality required. For example, the following scheme, when implemented on InterSystems DB Servers, will make the WebSocket server code work the same way as terminal based applications where *reads* are terminated by carriage return and *writes* immediately flush their data to the client.

First, adapt the client-side JavaScript code to append a carriage return character to each *send* operation.

```
function send(message) {
  if (ws) {
    ws.send(message + String.fromCharCode(13));
  }
}
```

Then the DB Server side code becomes:

```
MyWebSocket(%cgi,%content,%system)
     set %status=$$websocket^%zmgsis(.%system,0,"")
     new %system
     use 0:(/terminator=$char(13)::"+Q")
loop ; loop
     read data:30 if '$test write "Timeout at "_$H goto loop
     if data="exit" quit %status
     if data="$zv" s data=$zv
     write "DB server response at "_$H_": "_data
     goto loop
```

Note the simplified *read* operation and the absence of the need to *flush* the response.

7 License

Copyright (c) 2019-2021 M/Gateway Developments Ltd,
Surrey UK.
All rights reserved.

http://www.mgateway.com
Email: cmunt@mgateway.com

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.