Building a CGI Web Server

Martin Phillips, Technical Director, Ladybridge Systems

Overview

There are several excellent web development tools available for multi-value applications that enable rapid construction of web sites, however, at the time when we redeveloped the openqm.com web site to integrate it with our business systems, none provided all the features that we needed.

This presentation shows how it is possible to construct a dynamic web site using the Common Gateway Interface (CGI). Although the examples are mostly based on the openqm.com site and hence use of QM on Windows, the ideas discussed here are equally applicable to other multi-value database environments and operating systems. There are several code fragments to show how key elements of the system work but we have not reproduced the complete system here.

The main elements of the system are:

- A small CGI interface program written in C
- A transaction parser to decode incoming requests and assemble the response
- A menu display program that allows multi-level expanding menus
- An HTML display program that supports embedded control codes for dynamic data
- Application specific screen generation programs (nearly 100 in our complete business system)
- A library of HTML element generation subroutines

Aside from the application specific data files, the files that control the web page generation are:

- USERS User authentication
- SESSIONS Persistent data management
- HTML Template HTML pages
- MENUS Dynamic menu content
- LOG A transaction log to aid problem resolution

The CGI Program

There are several ways in which a web server can handle incoming traffic. One is for the URL to point to a CGI program that will programmatically construct a response based on data provided as part of the inbound message. In our case, the CGI program is written in C and simply calls a transaction parser subroutine written in QMBasic by using the QMClient API. Similar functionality can be achieved on UniVerse or Unidata using InterCall and on D3 using _CP_call.

The examples that follow are based on how the CGI program is used to handle the openqm.com web site though some of the site specific details have been changed. This application has a mix of traditional "text with hyperlink" pages and form filling. The elements of a form and other incoming data are passed to the server as an extension to the URL, for example,

```
mysite.com/cgi/cgi.exe?t0=m&t1=links&x=jlfo9d9pqn
```

where the data before the question mark is the web address of the CGI program and the data after the question mark is a list of parameter values with an ampersand (&) between each one.

A Windows version of the CGI program is reproduced below. The SERVER_xxx tokens should be replaced by appropriate values. Depending on how the browser sends the data, the parameters from the incoming web transaction are passed into the CGI program via an environment variable named QUERY_STRING or by reading a number of bytes specified in the CONTENT_LENGTH environment variable.

The QMCall() function returns the text of the web page to be displayed via its Response argument. If the page data is more than 32kb, the QMBasic part of the application writes the data to a temporary file and returns the pathname of this file prefixed by an exclamation mark. The C program detects this as a special case, emits the data to the web client and deletes the temporary file.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <io.h>
#include <\qmsys\syscom\qmclilib.h>
```

```
char InputData[32767] = "";       /* Incoming data */
char params[1000] = "";           /* CGI parameter names... */
char param_values[1000] = "";     /* ...and their values */
char Response[32767] = "";        /* Actual response or pathname of file */

void GetParam(char * name);
void CallServer(void);

/* ================================================================== */

int main()
{
 char * RequestMethod;  /* GET or POST */
 char buffer[1024];
 int bytes;
 int fu;
 char * p;

 RequestMethod = getenv("REQUEST_METHOD");
 if (RequestMethod == NULL)  /* Trap mis-use */
  {
   printf("Program must be executed by a Web browser\n");
   return 1;
  }

 /* Extract parameters that we might find useful in the server */

 GetParam("REMOTE_ADDR");    /* Client IP address */
 GetParam("HTTP_HOST");      /* Domain name used in url */

 if (!strcmp(RequestMethod,"GET"))
  {
   if ((p = getenv("QUERY_STRING")) != NULL) strcpy(InputData, p);
  }
 else if (!strcmp(RequestMethod,"POST"))
  {
   if ((p = getenv("CONTENT_LENGTH")) != NULL)
    {
     fread(InputData, atoi(p), 1, stdin);
    }
  }

 /* Pass request to database server program */

 if (!QMConnect(SERVER_ADDRESS, SERVER_PORT, SERVER_USER,
               SERVER_PASSWORD, SERVER_ACCOUNT))
  {
   strcpy(Response, "Failed to connect. The server may be offline.");
  }
 else
  {
   if (params[0] != '\0')
    {
     strcat(params, "\xfe");
     strcat(params, param_values);
    }

   QMCall("CGI", 3, InputData, params, Response);
   QMDisconnect();
  }

 /* Send response to client browser */

 printf("Content-type: text/html\n\n");
 printf("<meta http-equiv=\"Pragma\" content=\"no-cache\">\n");
 if (Response[0] == '!')   /* Large response - indirect via a file */
  {
   fu = open(Response+1, O_RDONLY);
   if (fu < 0)
    {
     printf("Internal error - Cannot open response file '%s' (%d)\n",
            Response+1, errno);
```

```
    }
   else
    {
     while((bytes = read(fu, buffer, 1024)) > 0)
      {
       fwrite(buffer, 1, bytes, stdout);
      }
     close(fu);
     remove(Response+1);
    }
  }
 else
  {
   printf("%s\n", Response);
  }

 return 0;
}

/* ====================================================================
   GetParam()  -  Build parameter list                                */

void GetParam(char * name)
{
 char * p;

 p = getenv(name);
 if (p != NULL)
  {
   if (params[0] != '\0')
    {
     strcat(params, "\xfd");
     strcat(param_values, "\xfd");
    }

   strcat(params, name);
   strcat(param_values, p);
  }
}
```

The Transaction Parser

The primary QMBasic component of this web interface is a catalogued subroutine declared as

```
subroutine cgi(input.data, params, reply)
```

The *input.data* argument receives the incoming web request, *params* is a multi-valued list of CGI related environment variable names and values as collected by the C program (only REMOTE_ADDR and HTTP_HOST in this example) and the *reply* argument is used to pass the constructed web page back to the CGI program.

We saw above that an extended URL might read

```
mysite.com/cgi/cgi.exe?t0=m&t1=links&x=jlfo9d9pqn
```

Each ampersand separated element of the text following the question mark represents a web transaction parameter. The naming of these is entirely controlled by the application and our usage is such that text fields are named as the case insensitive letter T followed by a number which identifies the field position in a dynamic array that will receive the associated text value. In the same way, we use B for buttons, C for checkboxes and R for radio buttons. The X item in this example is the session id which we will discuss later.

The process of parsing the parameters into the relevant dynamic arrays is simple except that it is necessary to decode some special HTML data constructs. Imaginatively, the T, B, C and R parameter items are parsed into common variable dynamic arrays named T, B, C and R. Perhaps such terse names are not a good idea but their use is well understood by the developers involved with this application.

```
   n = dcount(input.data, '&')
   for i = 1 to n
      s = field(input.data, '&', i)
      s[1,1] = upcase(s[1,1])  ;* Key should be case insensitive
      begin case
```

```
         case s matches '"B"1N0N"="0X'   ;* Command buttons
            j = matchfield(s, '"B"0N0X', 2)
            b<j> = parse(s, '"B"0N"="0X', 4)

         <<other cases for C, R and T go here>>
      end case
```

The parse() function is a local function that takes the same arguments as MATCHFIELD() but returns the text after processing any special embedded HTML constructs. It would be necessary to make some minor changes to convert this to an internal subroutine on systems that do not support local functions, returning the result value via a local variable.

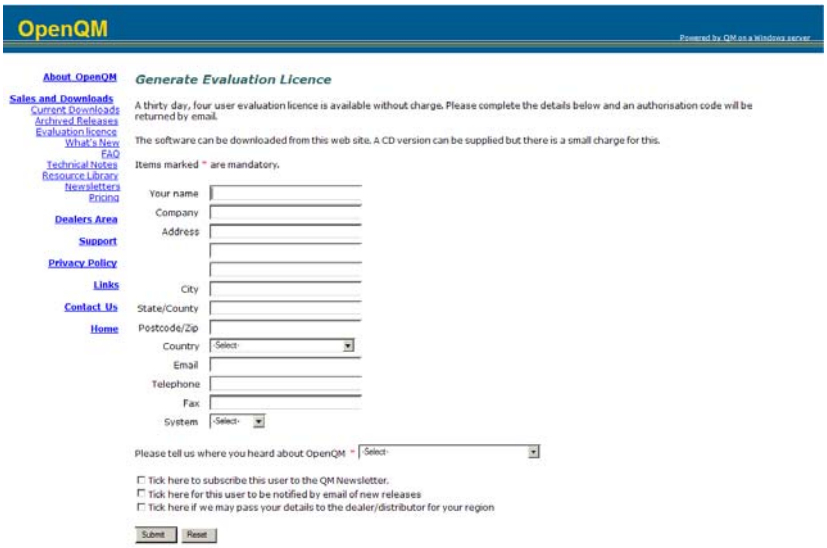
```
local function parse(indata, pattern, element)
   private s, k

   s = matchfield(indata, pattern, element)
   s = change(s, '+', ' ')
   for k = len(s) to 1 step -1
      if s[k,1] = '%' then
         s = s[1,k-1] : char(xtd(s[k+1,2])) : s[k+3,9999]
      end
   next k

   return trim(s, ' ', 'B')
end
```

To make things a little more complex, the parser used by the openqm.com web site also allows multi-valued tokens (for example, T5.2) but we will not need to look further at those here.

Because of the way in which this application evolved, there is an extra item, T0, which is not part of the T dynamic array but is used to identify the program that processes the screen or contains special values such as M for a menu action or H to display a pre-stored template HTML page.

The Page Layout

A typical page on the openqm.com web site might appear as



The page has three areas; a banner heading, a menu bar with expanding sub-menus, and a main page body. In this example, the body is a form to be completed by the user. For the purposes of this discussion we will consider the banner area as being constructed from fixed HTML text. The menu area and page body are generated by subroutine calls as discussed below. The transaction parser then merges these three area together to form the final text sent back to the browser.

Session Ids

Web transactions are inherently totally separate. There is no automatic persistence of data from one transaction to the next by the same user. There are various ways in which an application can provide its own persistence and the approach that we take here is to assign a unique id to the user's session which is then used as the record id to a SESSIONS file in which we can record whatever session related persistent data we need. Examples of such data for this application include the user's access level and details of the menus that are expanded on the menu bar.

When a user first connects, he will not have a session id and there will be no X parameter in the incoming data. In this case, the application creates a new session id from a random sequence of ten characters and writes a new session record for an unauthenticated user.

If a session id is provided in the incoming data, the application checks whether it is still valid. We timeout a session after a given period of inactivity, each new transaction for that session resetting the timer. There is also a mechanism to flush timed out sessions from the file.

Handling the Incoming Request

Once we have parsed the incoming data and, perhaps, linked it to an existing session, we are ready to process the action. The T0 parameter is used to identify what we are doing. For the purposes of this discussion we will look only at three codes.

A T0 value of M indicates a menu action where T1 identifies the menu name. If the session record shows that the named menu is not currently displayed, it is added to the list of menus to be expanded. Conversely, if it is currently displayed, it is removed from the list. Actual construction of the HTML elements to draw the menu is handled by a separate subroutine as described below.

A T0 value of H requests display of a template HTML page where T1 identifies the actual page to be displayed. These pages are stored in a file as simple HTML data but they can contain special substitution tokens that will get replaced before the data is emitted.

All other T0 values identify subroutines to be called to process the request. In each case, the subroutine name is formed by adding an internally defined prefix to the name in T0 so that it is only possible to call specific subroutines. The subroutine

builds an HTML data for the body of the page image in a common variable that will be merged with the rest of the page structure by the transaction parser.

Emitting the Basic Page Structure

After we have determined the menu bar content and built the page body, the time has come to emit the page structure. We do this simply by concatenating the HTML strings returned from the other parts of the system, inserting some fixed structures such as the HTML headers, style definitions and the page banner.

The final page text is then returned to the CGI program via the reply argument to the parser subroutine. If this text is over 32kb, it is written to a temporary file and the reply argument is set to the name of this file prefixed by an exclamation mark.

We will see how the main subroutine called from the CGI program assembles the page from components generated by other parts of the application.

Menus

The openqm.com web site uses a simple two level expanding menu structure though the underlying menu generation code can handle more than two levels. Clicking on a menu item that is itself an expanding menu will expand it if it is not already displayed or collapse it if it is displayed.

As well as changing the menu bar, it is sometimes useful for the main body of the page to change too. For example, clicking on the top level "Sales and downloads" menu item, expands the menu and changes the main body of the page to be a description of how to get a QM licence.

So, how does this work? It is actually a lot simpler than it might seem.

Menu definitions are stored in a MENUS file. Each entry has a multi-valued list of text items to appear on the menu and, associated with these, a multi-valued list of target item types (menu, HTML page, program, URL, etc), target names and access filters that control what class of user can see each item. There are also two single valued items that are handled in the transaction parser; an action relating to the main page body to be performed on expanding the menu and a similar action to be performed on collapsing the menu.

The program that handles menu display is shown below. This makes use of the ability for QMBasic programs to have recursively callable local subroutines that take arguments. A little reorganisation would be necessary on other systems to handle this with conventional internal subroutines.

```
function display.menu
$include common.h

   menu = '<br><div align="right"><font size="2">'

   gosub show(1, ses.rec<S.AREA>)
   menu := '</font></div>'

   return menu

* ======================================================================

   local subroutine show(depth, mnu.id)
      private mnu.rec, num.items, mnu.idx, text, type, action, filter

      read mnu.rec from mnu.f, upcase(mnu.id) then
         num.items = dcount(mnu.rec<M.TEXT>, @vm)
         for mnu.idx = 1 to num.items
            text = change(mnu.rec<M.TEXT, mnu.idx>, ' ', ' ')
            type = mnu.rec<M.TYPE, mnu.idx>
            action = mnu.rec<M.LINK, mnu.idx>
            filter = mnu.rec<M.FILTER, mnu.idx>

            if filter = '' or index(filter, ses.rec<S.LEVEL>, 1) then
               * Menu items

               begin case
                  case type = 'H'   ;* HTML document
```

```
                  menu := '<a href="':link('h','t1=':action):'">'
                  menu := if depth = 1 then '<b>':text:'</b>' else text
                  menu := '</a><br>'

               case type = 'M'   ;* Menu
                  menu := '<a href="':link('m','t1=':action):'">'
                  menu := if depth = 1 then '<b>':text:'</b>' else text
                  menu := '</a><br>'
                  locate upcase(action) in ses.rec<S.MENUS,1> setting pos then
                     gosub show(depth + 1, action)
                  end

               case type = 'P'   ;* Program
                  menu := '<a href="':link(action):'">'
                  menu := if depth = 1 then '<b>':text:'</b>' else text
                  menu := '</a><br>'

               case type = 'U'   ;* URL
                  menu := '<a href="http://':action:'">'
                  menu := if depth = 1 then '<b>':text:'</b>' else text
                  menu := '</a><br>'

               case 1
                  menu := text : '<br>'
            end case

            * If this is the top level menu, insert a blank line
            if depth = 1 then menu := '<br>'
         end
      next mnu.idx
   end

   return
end
end
```

Pre-Stored HTML Page Templates

Some pages of this web site require little more than display of pre-stored HTML. The mechanism used to handle these is more generalised and allows pages to be constructed from multiple pre-stored elements, either sequentially or nested. There are also special insertion tokens that can be embedded in the HTML to drop in variable data. For example, the openqm.com home page contains a reference to the current release number that is inserted totally automatically as the page is generated.

The template HTML text is stored in a file named HTML and is processed by a SHOW.HTML function. As this copies data from the stored record into the variable used to build the page, it looks for a number of special constructs.

A line commencing with a # is treated as a comment and totally ignored.

A line of the form

```
!name value
```

sets a variable that can be used later, typically in nested pages.

A line commencing with a question mark (?) controls conditional inclusion of what follows based on the class of the user displaying the page. For example, there might be a situation where a page should include text if the user is a dealer but not if they are a normal unauthenticated user. The form of this line is

`?IS` *xxx*	Includes following text only if the user is in class *xxx*
`?IS.NOT` *xxx*	Includes following text only if the user is not in class *xxx*
`?`	Includes following text unconditionally

The HTML data may also contain tokens that insert data from other sources. We chose to use the <<...>> delimiters that are also used for inline prompts in QM because HTML text never contains this sequence. Some of the more important insertion tokens supported by our system are

`<<CGI.LINK>>`	Inserts the URL of the CGI program
`<<HTML.xxx>>`	Insert HTML item *xxx*
`<<SESSION.ID>>`	Inserts the session id code
`<<TKN.`*xxx*`>>`	Insert item *xxx* from a control file
`<<`*name*`>>`	Insert token previously defined using !name

<<n>> Insert numbered argument from data supplied to the SHOW.HTML function

```
function show.html(page, args) var.args
$include common.h

   text = ''
   skipping = @false

   openseq 'HTML', upcase(page) to htm.f then
      loop
         readseq s from htm.f else exit

         c = s[1,1]
         if c = '#' then continue

         if c = '?' then   ;* Conditional element
            skipping = @false
            s = s[2,99999]
            operator = upcase(field(s, ' ', 1))
            begin case
               case operator = 'IS' or operator = 'IS.NOT'
                  classes = upcase(field(s, ' ', 2))
                  skipping = (index(classes, user.type, 1) # (operator = 'IS'))
            end case
            continue
         end

         if skipping then continue

         if c = '!' then
            token.name = upcase(trim(field(s[2,99999], ' ', 1)))
            locate token.name in html.token.names<1> setting pos else
               html.token.names<pos> = token.name
            end
            html.token.data<pos> = trim(field(s, ' ', 2, 999))
            continue
         end

         c = s[1]
         if c # ' ' and c # '>' then s := ' '

         * Add session id into any href tokens

         s = change(s, 'href="?', 'href="?X=':session.id:'&')

         * Look for substitution tokens

        s2 = ''
        loop
        while s matches '0X"<<"0X">>"0X'
           s2 := matchfield(s, '0X"<<"0X">>"0X', 1)
           token = upcase(matchfield(s, '0X"<<"0X">>"0X', 3))
           s = matchfield(s, '0X"<<"0X">>"0X', 5)

           begin case
              case token matches '1N0N'
                 s2 := args<token>
              case token = 'CGI.LINK'
                 s2 := cgi.link
              case token[1,5] = 'HTML.'
                 s2 := show.html(token[6,999])
              case token = 'SESSION.ID'
                 s2 := session.id
              case 1
                 locate token in html.token.names<1> setting pos then
                    s2 := html.token.data<pos>
                 end else
                    read ctl.rec from ctl.f, 'TKN.':token then
                       s2 := ctl.rec
                    end
               end
           end case
        repeat
```

```
        text := s2 : s : char(10)
      repeat
   end else
      text = '<p>Cannot find HTML document "':page:'"</p>'
   end

   return text
end
```

Screen Handling Programs

The bulk of the facilities offered by the openqm.com web site are provided via a set of nearly 100 screen programs that display or input data, usually in the form of HTML tables.

The T0 parameter identifies the program to be executed. For example, setting T0 to "login" runs the program responsible for user name and password authentication for the secure areas of the site. Aside from the application logic, the web page aspects of this program become little more than a series of calls to a library of functions that generate each of the data elements to appear on the screen. For example, display of the page that allows dealers to authenticate their login credentials is shown below.

```
body = title('Dealers Area Login')
body := form('login2')

body := '<p>Please login for access to the dealers area of this site.</p>'
body := '<table rules="none">'
body := table.entry(1, 'User name', @true)
body := table.entry(2, 'Password||password', @true)
body := '</table><br>'

body := button(1, 'Login')
body := '</form>'
body := set.focus('T1')
```

Many operations require a sequence of screens. Rather than these becoming separate programs, multi-screen sequences are handled by adding a numeric suffix to the screen program name in the T0 variable. This is stripped out by the input data parser and stored in a variable that identifies the screen number in the sequence. Thus, for example, generation of an evaluation licence starts with T0 set to "generate" and then goes on to "generate2", "generate3" and so on. Because it would be possible for a user to construct a URL that dived in part way through a screen sequence, the program performs various integrity checks along the way.

HTML Generation Functions

Several of the code fragments we have seen above have referenced functions that generate HTML elements. We have a library of about twenty of these that allow construction of display tables, data entry tables, check boxes, drop down selection lists, etc. For example, a slightly simplified version of the function that emits a table element containing an input text box is shown below.

```
function table.entry(idx, text)
$include common.h
   s = '<tr><td align="right">' : text : ' </td>'
   s := '<td align="left"> '
   s := '<input type="text" name="T':idx:'"'
   if t.err<idx> then s := ' style="background: lightsalmon"'
   s := ' size="35" value="':t<idx>:'"/>' : '</td></tr>'

   return (s)
end
```

This function shows that highlighting of errors is controlled by a dynamic array named T.ERR with a field by field correspondence to the T dynamic array that holds the input data. There are similar error arrays for the other input elements.

Security

This application has four distinct levels of access. The session record holds the access level associated with the session. The menu building elements of the application attach an access level to each menu item and only display it where it is valid. Again, because a user could construct a URL of his own in an attempt to bypass this mechanism, every program begins with a call to a function that checks if the user really should be allowed in, displaying an error at a security violation.