Table of Contents

Change History

- August 2004 – Created this document
- March 2008 – Converted from a web page booklet to a PDF file, using cut/paste mainly so formatting is a bit whacky
- January 2009 – Figured out what header value sets cookies so replaces the section on how to do it in JavaScript with how to do it by writing the header values. Also removed the sample cgi library and sample application sections as they are not relevant to the tutorial and had become outdated.
- May 2012 – updates to clarify some of the information and provide some example web scripts
- Dec 2019 – updated to include the link to the wikipedia entry on all available header fields as section 7 as I had a need to set the http response code and other users may be interested in all the options available

Redistribution of this document

1. Overview - What is this book and who is it for.

What is this book ?

This book is primarily designed to show that you don't need to learn to program in new languages to write a web based application using cgi scripting tools. It will show that you don't need perl, php, java etc to write a web based application.

It will also show you that you don't need a book. Everything in this reference is simple, straight-forward, and easy to follow and use.

This book will take you through how to use nothing but standard shell scripting practises to write a full function web application. It will show you how to handle http GET and POST requests, and how to manage cookies; all from simple shell scripts.

It is written for the Apache web server running in a Unix (Linux) environment.

The tutorial scripts here hava also been tested under the IBM implementation of the http server, which behaves differently to the apache web server in respect of some environment variables; so only common variables are discussed.

This is not a reference on the HTTP protocol. I make mention of needing two blank lines between the content-type and html content if you are not using additional http headers, and highlight that you only need one blank line between any supplied http headers if they are present (in the cookie section); but this book is not going into the explanations of why.

It is a specifically a how-to, not a why you do.

Who is it for ?

Systems Admins and Unix nuts
My first home web server did everything using PERL as that was what all the books were teaching at the time. Get a job in the real world and very few of the servers run PERL (or web servers but thats irrelevant). After 3-4 years the PERL knowledge starts to fade away, but shell scripts are written every day, why not use them for web apps ?

Possibly because there are books on using PERL for CGI scripting, but not a lot around for native shell scripting. So there was a need for one, and time for a free one !.

This is for anyone that can write shell scripts in their sleep, but does not want to learn any other languages in order to have full function applications running on their personal web sites.

Disclaimer

Authors disclaimer: Don't use any form of CGI application on a publily accessble network (perl, ksh, csh, bash etc) as you will be begging for trouble. Keep CGI scripting inside internal networks.

2. Standard Web Server Variables Available.

What are the standard environment variables

Every Web Server provides a set of standard environment variables that you can use to obtain information about the web server that is running your script, and some basic information about the remote computer requesting a page from your web server. These variables are always available.

These environment variables can be used at the start of your scripts to perform basic environment or security checks. For example you can ensure the request came from the internal network and thow up an error or e-mail the sysadmin if not.

Whats covered here ?

This section covers the key standard environment variables. To see how to obtain additional information that could be provided if the web page making the request contains it (forms, buttons etc) see the next chapter.

Standard Environment Variables

Different vendors Web servers provide different environment variables, although that is mainly due to default security changes; for example the default IBM http server while based on apache will provide less environment variables than a standard default apache install. Customisation also plays a part as I have customised my apache installation to also suppress some variables and many other sites will also do so.

The easiest way for you to see all the standard default environment variables that your web server currently provides is to write a cgi script as in the sample below, place it in your cgi-bin directory, and load the page.

Sample cgi script to show environment variables

```
#!/bin/bash
echo "Content-Type: text/html"
echo ""
echo ""
echo "<html><head></head><body><pre>"
env
echo "</pre></body></html>
exit 0
```

The results of the above script run on my Linux Fedora Core 16 Apache Web Server yields the following environment variables. The highlighted ones are the only ones you need to care about for using scripts as a web application; the exception being cookies that also set environment variables but cookies are covered separately in later sections.

```
SERVER_SIGNATURE=
PERL5LIB=/usr/share/awstats/lib:/usr/share/awstats/plugins
HTTP_USER_AGENT=Mozilla/5.0 (X11; Linux i686; rv:12.0) Gecko/20100101
Firefox/12.0
SERVER_PORT=80
HTTP_HOST=falcon
DOCUMENT_ROOT=/home/httpd/playpen/html
SCRIPT_FILENAME=/home/httpd/newsite/cgi-bin/testdoc/show_env.sh
REQUEST_URI=/cgi-bin/testdoc/show_env.sh
SCRIPT_NAME=/cgi-bin/testdoc/show_env.sh
HTTP_CONNECTION=keep-alive
REMOTE_PORT=4907
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/home/httpd/newsite/cgi-bin/testdoc
SERVER_ADMIN=mark@localhost
APPLICATION_ENV=production
HTTP_ACCEPT_LANGUAGE=en-us,en;q=0.5
HTTP_DNT=1
HTTP_ACCEPT=text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
REMOTE_ADDR=192.168.1.187
SHLVL=1
SERVER_NAME=falcon
SERVER_SOFTWARE=Apache/2.2.22 (Fedora)
QUERY_STRING=
SERVER_ADDR=192.168.1.183
GATEWAY_INTERFACE=CGI/1.1
SERVER_PROTOCOL=HTTP/1.1
HTTP_ACCEPT_ENCODING=gzip, deflate
REQUEST_METHOD=GET
_=/usr/bin/env
```

Discussion on Variables Usefull in scripts

Here we briefly touch on the standard environment variables from the table above **that will be usefull to you in your scripts**. Where I am aware of any issues with using these variables there are appropriate notes.

REMOTE_ADDR	The address of the client that is accessing the site. You could use this at the start of shell scripts as a security check to ensure only local or known address can continue running the script. Can also be used for tracking if you want to.
QUERY_STRING	This is the query part of the URL if the script was called with query parameters in the form http://cgi-bin/scriptname.sh?aaa=bbb&ccc=ddd ***as is done in a form that uses the GET method*** (or as links you may imbed in your pages). It is a single variable with all the parameters passed as part of the URL used. **Tip:** the QUERY_STRING variable is available for both GET and POST requests, so if you really wanted you could also pass query string information in the same http regest as a form using the POST method using something like this `<form action=http://xxx.sh?aaa=bbb&ccc=ddd method=POST>` `<input type=hidden name=fred value=barney>` `</form>` and in your script parse the POST data for the 'name' and the QUERY_STRING for 'aaa' and 'bbb'. Probably not much real use but a usefull trick on occasion; as long as your processing script expects data in the query string as the request method will be POST.
REQUEST URI SCRIPT_NAME	This is the name of the script as known to the web server. Similar to SCRIPT_FILENAME but with the path the web server knows it as (*relative to the site*)
SERVER_ADMIN	This is the e-mail address defined in the web server configuration file as belonging to the server administrator(s). This will be usefull if you have a contact admin page or want to put a mail link to contact the admin at the base of your web pages.
REQUEST_METHOD	The only two values we care about for script applications are GET or POST to process form input. There can be other values, such as 'connect' but if you see any of those (or anything other than get or post) check your firewall logs.

Chapter Summary

There are a set of standard environment variables made available to any script by the web server. What variables are provided is web server dependant, but those identified here can be usefull in scripts.

3. Important note on script output requirements

The sample script in section 2 to show the environment variables gives an example, but the rules that must be followed are

- You must always provide the content-type
- There must always two blank lines between the content type and html tag (unless you are passing additional header information to the web server as discussed in the cookie section; but generally you will have two blank lines)

```
For example
      Content-Type: text/html


      <html><head></head><body>
      <!== your web page content, scripts, *nix commands etc here ==>
      <!== end of your content ==>
      </body></html>
```

Failure to follow those rules will result in a blank web page being displayed to the web browser with no errors being logged anywhere for you to debug.

Note: any script errors may also result in an incomplete or partial web page being displayed to the web browser, but script errors can generally be found recorded in the apache error_log file.

4. Obtaining Information Entered in Forms.

Obviously to write an application using shell scripts you will need to be able to process input the user enters into forms. Yes, it's simple.

An HTML form provides two methods of sending data to a web server, the GET method and the POST method. Both provide data in different ways.

Forms using the GET method

The GET method is intended for forms *that need to send only a small amount of data* to the web server. It does this by appending the data to the end of the URL used to refer to the script.

- The seperator between the URL itself and the trailing data component is the **?** (question mark) character.
- Within the data values each data element is seperated by the **&** (amphersand) symbol. The data elements are in the form name=value.

Assume for example you have a html form with a method=GET and have the fields **name** and **age** where the user has entered "fred bloggs" and "31" respectively. If the action was "/cgi-bin/test.sh" then the resulting URL used to trigger your script would be

http://<server-dir>/cgi-bin/test.sh**?**name=fred%20bloggs&age=31

Whats the %20 you ask, thats the space. All special characters are translated to %nn values somewhere between the client browser, the web server, and our script; where exactly ?, we don't need to care for the purposes of writing scripts. We just need to file this away and remember to handle it.

For the purposes of accessing the data provided you should use the web server created environment variable **$QUERY_STRING**. This saves you having to parse up to the ? at least. This will contain everything after the ? giving you a little less work. You will need to parse this in your scripts to split out the field names and field values.

Note that keyword=value pairs may be provided in any order, that is decided by the browser used and not by the order you put fields onto a form, you scripts must handle that.

UPDATE 19May2012: While rewriting this I have noticed that FireFox12 passes a space as + rather than %20 (similar to how a post request passes a space); not sure when that changed in FireFox or the apache web server so you will have to allow for both %20 and + to be used as space characters when your scripts process a get request. The same script in 2004 used a %20. Run the script in your environment to see which is relevant to you.

Processing the data provided by a GET method

How you process these fields is up to you. Basically you need to identify all the individual data elements by parsing for the **&** (amphersand) character to get all your name=value fields, then parse each of those to obtain the values for each name.

Quite honestly you are in for a lot of 'awk' and 'cut' forking here it you are not using the **bash** shell. **bash**, bless it's little heart provides an inbuilt substring function so chomping through these fields is easy; if you use sh or ksh you are going to be stuck writing code like that in the sample here.

If you know exactly how many fields you are going to be recieving you could do something like the below using the previous example of name and age fields, knowing there were only two fields (remembering **they can come in any order** so you can't even assume with only two fields they will be in the order you put them on the form)

The example script below will just throw up a input form, and on a refresh show the values entered into the input fields. *On your own test server change the action to where you put the script of course*.

As you can see this (the field parsing) is rather messy. The example I've given here though messy is the sort of code I have to write on servers without bash.
The script will display a form to input name and age into, and send the data to itself to to display.

```
#!/bin/bash
# ----------------------------------------------------------------
# Ok not a pretty form, its just an example showing GET processing
#
# (1) display webpage with form input area, submit button calls
#     this script again
# (2) if any query string (form submit to refresh page) was
#     provided show the data entered below the input form
#
# No validation of the data entered is done.
#
# NOTES
# Wherever possible limit the size of your form input fields
# as done here. While that cannot remove the need to sanitise
# input on the server side to prevent buffer overruns it
# certainly doesn't hurt to have limits in the form (maxlength).
# ----------------------------------------------------------------
cat << EOF
Content-Type: text/html


<html><head><title>Test GET Form processing</title></head><body>
<h1>Data input area</h1>
<!== change the action to where you put the test script ==>
<form method="get" action="/cgi-bin/testdoc/getform_demo.sh">
Name: <input name="username" type="textarea" size="30" width="30"
maxlength="30" value="">
Age: <input name="age" type="textarea" size="5" width="5" maxlength="5"
value="">
<input type=submit value="Submit Data"><input type=reset>
</form>
<br />
<h1>Data output area</h1>
EOF

# ------------------------------------------------
# now see if anything to display from query string
# ------------------------------------------------
if [ "${QUERY_STRING}." != "." ];
then
   field1=`echo "${QUERY_STRING}" | awk -F\& {'print $1'}`
   field2=`echo "${QUERY_STRING}" | awk -F\& {'print $2'}`
   # Whats field 1
   fieldname=`echo "${field1}" | awk -F\= {'print $1'}`
   fieldval=`echo "${field1}" | awk -F\= {'print $2'}`
   if [ "${fieldname}." = "username." ];
   then
     name="${fieldval}"
   fi
   if [ "${fname}." = "age." ];
   then
     age="${fval}"
   fi

   # Whats field 2
   fieldname=`echo "${field2}" | awk -F\= {'print $1'}`
   fieldval=`echo "${field2}" | awk -F\= {'print $2'}`
   if [ "${fieldname}." = "username." ];
   then
     name="${fieldval}"
   fi
   if [ "${fieldname}." = "age." ];
   then
     age="${fieldval}"
   fi

   # Always sanitise, or in this demo just check, input values
   if [ "${name}." = "." -o "${age}." = "." ];
   then
      echo "Waiting for data entry in both fields<br />"
   else
      echo "Results:<br />Name: ${name}<br />Age: ${age}<br />"
   fi
else
   echo "No query string to be processed yet."
fi
echo "</body></html>"
exit 0
```

Additional Considerations

A timely reminder, don't forget that data passed to the web server and made available to your scripts has had all the 'special characters' changes to %nn text rather than the character that was origionaly present (or space to + instead of %20 depending upon your browser implementation).

Your script must translate these % fields back to the original characters where appropriate (using a bit of common sense of course, I ALWAYS translate the character for ` (exec) to null as I don't want any scripts accidentally running commands a user may enter).

Or in english, in real life if you are using shell scripts you should dump $QUERY_STRING through a sed pipe to remove any dangerous chatacters rather that assume it is safe.

Forms using the POST method

The POST method is used for forms that need to provide a lot more data than is permitted in a URL string. The web server will provide that to the script through the standard input (STDIN). The data provided is similar to that of the query string discussed earlier where there are name=value pairs seperated by the & (amphersand) character.

One important difference that you must be aware of is that in POST data you will find that spaces are sent in the data stream as the + (plus) character instead of the %20 used in a GET data stream.

This method of passing the data through standard input (stdin) to scripts is probably why scripts should not be used on publicly accessable networks as there is a good chance of a buffer overflow. To read the data POSTed you must put the entire data stream into a single variable and hope your shell language (and server) can buffer it.

To read POST data you must do something like "**databuffer=`cat`**" and hope the shell can buffer it.

So if you are processing a POST type form using scripts ensure any form you provide to the user has a limit on the size of data that can be entered (that won't stop anyone taking a copy of your form and removing the limit; so reminding you yet again never use a shell scripted application on a public network).

You will need to parse the variable the data is dumped into in your scripts to split out the field names and field values.

Note that keyword and value pairs may be provided in any order, that is decided by the browser used and not by the order you put fields onto a form, you scripts must handle that.

Processing data provided by a POST method

I personally have never been able to cat the 'stdin' stream into a processing loop, stuffing into a variable buffer is the only way I have ever been able to use it, other *nixes may allow it but in order to stay portable I'd suggest staying with dumping it into a variable and making sure you have all the latest patches to prevent shell variable buffer overruns.

How you process these fields is up to you. If you have bash, then good. If you are trying to use sh or ksh you are in for a lot of 'awk' and 'cat' forks to extract these fields.

You will need to parse out all pairs by splitting on the **&** (amphersand) symbol, and then store in your code the fieldname and field values. If you know exactly how many fields you are going to be recieving you could do something like the below using the previous example of name and age fields, knowing there were only two fields (remembering **they can come in any order**).

As you can see this is rather messy. The example I've given here though messy is the sort of code I have to write on servers without bash. Actually, in all honesty, there are a lot of awk calls even with bash, but the bash inbuilt substring function takes all the guesswork out of determining how many fields were provided.
And you will note it is almost identical to the example for processing data sent by the GET method, the only difference being that with the POST method data is provided via STDIN rather than an environment variable.

```
#!/bin/bash
# -------------------------------------------------------------
# Ok not a pretty form, its just an example showing POST processing
# (1) display webpage with form input area, submit button calls
#     this script again
# (2) if any query string (form submit to refresh page) was
#     provided show the data entered below the input form
#
```

```
# No validation of the data entered is done.
#
# NOTES
# Wherever possible limit the size of your form input fields
# as done here. While that cannot remove the need to sanitise
# input on the server side to prevent buffer overruns it
# certainly doesn't hurt to have limits in the form (maxlength).
# -----------------------------------------------------------------
cat << EOF
Content-Type: text/html


<html><head><title>Test POST Form processing</title></head><body>
<h1>Data input area</h1>
<form method="post" action="/cgi-bin/testdoc/postform_demo.sh">
Name: <input name="username" type="textarea" size="30" width="30"
maxlength="30" value="">
Age: <input name="age" type="textarea" size="5" width="5" maxlength="5"
value="">
<input type=submit value="Submit Data"><input type=reset>
</form>
<br />
<h1>Data output area</h1>
EOF

# -------------------------------------------------
# now see if anything to display from query string
# -------------------------------------------------
# get the data buffer
xxx=`cat`
if [ "${xxx}." != "." ];
then
   field1=`echo "${xxx}" | awk -F\& {'print $1'}`
   field2=`echo "${xxx}" | awk -F\& {'print $2'}`
   # Whats field 1
   fieldname=`echo "${field1}" | awk -F\= {'print $1'}`
   fieldval=`echo "${field1}" | awk -F\= {'print $2'}`
   if [ "${fieldname}." = "username." ];
   then
     name="${fieldval}"
   fi
   if [ "${fname}." = "age." ];
   then
     age="${fval}"
   fi
   # Whats field 2
   fieldname=`echo "${field2}" | awk -F\= {'print $1'}`
   fieldval=`echo "${field2}" | awk -F\= {'print $2'}`
   if [ "${fieldname}." = "username." ];
   then
     name="${fieldval}"
   fi
   if [ "${fieldname}." = "age." ];
   then
     age="${fieldval}"
   fi
   # Always sanitise, or in this demo just check, input values
   if [ "${name}." = "." -o "${age}." = "." ];
   then
      echo "Waiting for data entry in both fields<br />"
   else
      echo "Results:<br />Name: ${name}<br />Age: ${age}<br />"
   fi
```

```
else
   echo "No data entered to process yet"
fi
echo "</body></html>"
exit 0
```

Additional Considerations

A timely reminder, don't forget that data passed to the web server and made available to your scripts has had all the 'special characters' changes to %nn text rather than the character that was origionaly present. Your script must translate these % fields back to the original characters where appropriate. **An additional consideration** is that POST data fields use the **+** (plus) symbol to indicate a space rather than the %20 traditionally used by a GET request.

5. How to Manage Cookies.

Cookie Considerations

As a general rule Cookies are provided from the client to the server if the script is; either running in the same, or under, the directory of the script that set the cookie in the first place.

/cgi-bin/scriptdir1/set_cookie.sh	Creates the cookie using default path
/cgi-bin/scriptdir1/read_cookie.sh	OK, cookie will be sent to server
/cgi-bin/scriptdir1/nextdir/read_cookie.sh	OK, cookie will be sent to the server
/cgi-bin/read_cookie.sh	Cookie is not sent, not the full path
/cgi-bin/scriptdir2/read_cookie.sh	Cookie is not sent, not the full path

If you want scripts that are not in the current script path to be able to abtain cookies you are setting you must provide a root directory name for the application using the cookie that must be high enough up the URL path to ensure it encompasses all your scripts.

So if your script was expecting to get a cookie and it didn't get it, before running around checking all your script logic check your script paths to ensure the script was entitled to get it.

How to read cookie Values

All cookie values for the appropriate page being processed are automatically sent from the client to the server; this is handled between the browser and the web server so you don't need to worry about it. All you need to worry about is how to read a cookie value.

This is relatively simple from shell scripts. The web server will place the contents of all the cookies it has been sent by the client that are relevant to the page being processed into another environment variable. This time the environment variable is named **HTTP_COOKIE**.

The format of the data string here is value=name as for the other environment variables set by the server except for a change in seperator character. The seperator character between cookie values is the **;** (semi-colon) followed by a space rather than the & that is used everywhere else.

For exmaple two cookies "test1=value 1" and "test2=value 2" will be returned in the HTTP_COOKIE environment variable as
test1=value%201; test2=value%202

It is **also important** to note that the cookie names and values will have special characters translated to %nn as was done for the GET and POST requests, hence the %20's replace the spaces in the example above. You will have to translate these back to real characters.

UPDATE 19May2012: while updating this I have noticed (using the test script) that FireFox12 is now passing spaces in cookie values as actual spaces. Whether this is a change in FireFox or in the apache server that is allowing them to be shown as spaces I don't know. The same script showed %20 in 2004. Run the script in your environment to see what is relevant to you.

How to write cookie values

There are a couple of ways to do this. You could manually write the values into the http header, or you could 'echo' javascript into the page sent to the client to do so.

As this tutorial is not about javascript I will just cover how to do it by writing the values directly to the http response header, which is also a lot easier.

The only five key things you need to remember are

- The field values on the server are seperated by the ; character
- The syntax: Set-Cookie: NAME=VALUE; expires=DATE; path=PATH;domain=DOMAIN;secure
- This must be set in the response header, **before any HTML is sent**
- Only the first field is actually required. If no expiration is given the cookie is deleted when the user exits the browser
- To delete a cookie, set the expiry date to any time in the past and the client browser will (should if it is standards compliant, some are not) delete the cookie

This is a simple example showing how to set to cookies in the users browser client.
The first time the cgi-script is run there will be nothing output from the HTTP_COOKIE value, as it is not until the browser gets the response page it sets the cookies.
The next time the cgi-script is run the response page will contain the two cookie values as they will be sent to the server (now they have been set from the first request)

```
#!/bin/bash
# An example of setting cookie values using a shell script.
#
# The first time the browser calls this script there will
# be nothing displayed as a cookie value, as it is not until
# the browser retrieves the page it will get the request
# to set cookies (so it had no cookies to pass to the server)
#
# When you refresh the page the cookies that were set will
# be sent to the server along with the GET page request
# so will be available to be displayed in the response page.
cat << EOF
Content-Type: text/html
Set-Cookie: MyCookie1=TEST DATA 1; expires Wed, 13 Jan 2009 12:00:59 GMT
Set-Cookie: MyCookie2=TEST DATA 2; expires Thu, 14 Jan 2009 17:00:00 GMT

<html>
<head><title>Cookie Test</title></head>
<body>
Cookie Values Sent from the web page to the Server:<br /><br />
EOF
echo "${HTTP_COOKIE}"
echo "</body></html>"
exit 0
```

Important note: As you will see by the example now we are providing additional header information in the data stream sent to the browser only one blank line is needed between the headers and the html content.

So now you know how easy it is to set Cookies, well maybe not quite... using javascript or PHP functions you can set the expiry using inbuilt date functions (ie: in 30 mins time) which is a bit difficult to do and get the correct format in a shell script, but certainly possible.

Also the cookies will persist for the life of the browser session reguardless of the expiry date you set.

6. Wrapping it all up

A simple way of handling both GET and POST requests

You can write a complete application using just shell scripts from both GET and POST form requests simply by understanding GET requests are stored in the QUERY_STRING environment variable and POST requests have data piped through standard input.

Something like the below would handle both cases.

```
   case "${REQUEST_METHOD}" in
      "POST") HTTPDATABUFFER=`cat` ;;
      "GET")  HTTPDATABUFFER="${QUERY_STRING}" ;;
      *)      exit 1    # ohh, we don't cover other options,
                        # you are probably being hacked,
                        # check you apache logs for things
                        # like 'connect' where you are being
                        # targted as a relay/proxy
              ;;
   esac
  # do something with the data in HTTPDATABUFFER now
```

But you should not do that !.
Your application should know if it expects POST or GET and as shell scripts are not the safest CGI interface you should treat data provided in an unexpected way as a hacking attempt.

A working example of that

This script is a quick script example using the method shown above to process both GET and POST data input.

The script will alternate between using the GET and POST methods each time the firm is submitted to the server.

And because we can a cookie will be set showing the action method used the last time the form was submitted.

```
 #!/bin/bash
# ************************************************************
# Throwing it all together
#
# - form starts (requested via get) off setting the imput form
#   to use the post action
# - each time the form is submitted it switched between post
#   and get action modes
# - the cookie vaule is set to form action type so after submit
#   is used the cookie value shown on the output area will be
#   the action type of the page calling the current page (the
#   previous mode)
# = no translation of the input data fields is done to change
#   special character fields back to the origional values,
#   as this is not a sed tutorial and it will mess up the
#   example code.
#
# As I am not parsing the input I see no need to do that
# before displaying the form, and putting the values back
# into the input fields when the form is displayed.
# You can play with that as an exercise.
# **********************************************************
# Initialise variables
HTTPDATABUFFER=""
name=""
age=""
name_error=""
age_error=""

# ----------------------------------------------------------
# throw_unsupported_page:
#   Throw up this page if the script was not called with
#   action type get or post.
# ----------------------------------------------------------
throw_unsupported_page() {
method="$1"
cat << EOF
Content-Type: text/html


<html><head><title>Error</title></head><body>
Page was called with REQUEST_METHOD=${method}<br />
That is NOT supported by this example.
</body></html>
EOF
exit 0
} # end throw unsupported_page

# ----------------------------------------------------------
# buffer_translate:
#   convert the %nn characters back to real characters for
#   display, remove any nasty ones
# DEMO ONLY: there are a lot more characters you may want to
#           translate later; and in the real world you
#           would want to do the whole lot on one sed command
#           instead of multiple as I have shown here. I just
#           wanted to throw in a few comments here.
# ----------------------------------------------------------
buffer_translate() {
  databuffer="$1"
# Firefox12 at least passed a space as a + instead of a %20 now
# or this is something done by a later version of apache than
# used by my origional doc.
# Translate + to space for get requests as well, untested
# on what effect that will have on IE6 or earlier apache
# versions.
# rather than by a browser
#   if [ "${REQUEST_METHOD}." = "POST." ];
#   then
      databuffer=`echo "${databuffer}" | sed s/+/\ /g`
#   fi
   databuffer=`echo "${databuffer}" | sed s/%20/\ /g`
   databuffer=`echo "${databuffer}" | sed s/%2C/,/g`
   databuffer=`echo "${databuffer}" | sed s/%28/\(/g`
   databuffer=`echo "${databuffer}" | sed s/%29/\)/g`
   databuffer=`echo "${databuffer}" | sed s/%3D/\=/g`
   databuffer=`echo "${databuffer}" | sed s/%60//g`   # drop `
   databuffer=`echo "${databuffer}" | sed s/%40/@/g`
   databuffer=`echo "${databuffer}" | sed s/%3F/\?/g`
   databuffer=`echo "${databuffer}" | sed s/%22//g`
   databuffer=`echo "${databuffer}" | sed s/%27/\'/g`    # single quote, drop if not needed

   # next two are together, assuming output is to be displayed on a web page
   databuffer=`echo "${databuffer}" | sed s/%0D//g`     # NL and LF to just be html line break
   databuffer=`echo "${databuffer}" | sed s/%0A/\<br\>/g`

   databuffer=`echo "${databuffer}" | sed s/%7E/~/g`
   databuffer=`echo "${databuffer}" | sed s/%21/!/g`
   databuffer=`echo "${databuffer}" | sed s/%23/#/g`
   databuffer=`echo "${databuffer}" | sed s/%24/$/g`
   databuffer=`echo "${databuffer}" | sed s/%25/%/g`
   databuffer=`echo "${databuffer}" | sed s/%5E/^/g`
   databuffer=`echo "${databuffer}" | sed s/%26/\&/g`
   databuffer=`echo "${databuffer}" | sed s/%2B/+/g`
   databuffer=`echo "${databuffer}" | sed s/%3A/:/g`
   databuffer=`echo "${databuffer}" | sed s/%3B/\;/g`
   # The / character is a pain, fiddle with it to a ~ then tr it as below
   databuffer=`echo "${databuffer}" | sed s/%2F/~/g`
   databuffer=`echo "${databuffer}" | tr "~" "/"`
   echo "${databuffer}"
} # end buffer_translate

# ----------------------------------------------------------
# extract_fields:
#   get the two values we want
#   in a seperate routine as I decided for this demo that the
#   data provided should also be put back in the form fields
#   when the form is reposted, so it had to be done before
#   the actual form is redisplayed.
# ----------------------------------------------------------
extract_fields() {
   if [ "${HTTPDATABUFFER}." != "." ];
   then
      field1=`echo "${HTTPDATABUFFER}" | awk -F\& {'print $1'}`
      field2=`echo "${HTTPDATABUFFER}" | awk -F\& {'print $2'}`
      # Whats field 1
      fieldname=`echo "${field1}" | awk -F\= {'print $1'}`
      fieldval=`echo "${field1}" | awk -F\= {'print $2'}`
      if [ "${fieldname}." = "username." ];
      then
        name="${fieldval}"
      fi
      if [ "${fname}." = "age." ];
      then
        age="${fval}"
      fi
      # Whats field 2
      fieldname=`echo "${field2}" | awk -F\= {'print $1'}`
      fieldval=`echo "${field2}" | awk -F\= {'print $2'}`
      if [ "${fieldname}." = "username." ];
      then
        name="${fieldval}"
      fi
      if [ "${fieldname}." = "age." ];
      then
        age="${fieldval}"
      fi
   fi
   # sanitise the data for display (translate special chars to real chars)
   if [ "${name}." != "." ];
   then
      name=`buffer_translate "${name}"`
   else
      name_error="No name was provided to the form"
   fi
   if [ "${age}." != "." ];
   then
      age=`buffer_translate "${age}"`
      # age must be a number
      if [[ "$age" != ?(+|-)+([0-9]) ]];
      then
         age_error="age value was not numeric (${age})"
         age=""
      fi
   else
      age_error="No age was provided"
   fi
} # extract_fields

# ---------------------------------------------------------
# show_entered_data:
#   display the submitted data
#   extract the two field values from the sumbitted data
#   display the two field values
#   display the cookie value
# ---------------------------------------------------------
show_entered_data() {
   echo "<h1>Data output area</h1>"
   cat << EOF
```

```
name: ${name} ${name_error}<br />
age:  ${age} ${age_error}<br />
cookie: ${HTTP_COOKIE}
EOF
} # end show_entered_data

# ----------------------------------------------------------
# display_form:
#   display the input form. The action type will be set to
#   get or post based on the parameter passed.
# ----------------------------------------------------------
display_form() {
  usetype="$1"
  cat << EOF
Content-Type: text/html
Set-Cookie: TestType=${usetype}; expires Wed, 13 Jan 2009 12:00:59 GMT

<html><head><title>Test ${usetype} Form processing</title></head><body>
<h1>Demo bash script page</h1>
<p>
A sample bash script showing how to process form input using both
the GET and POST methods, plus setting and retrieving cookies.
</p>
<p>
This test form will set a cookie indicating the type of processing
being used (get or post as shown above), and when the form is sumbitted
it will switch between get/post modes and the cookie displayed in the
results will show the mode of the previous mode.
</p>
<p>The current form will use <b>${usetype}</b> processing.</p>
<h1>Data input area</h1>
<!== change the action to where you put the test script ==>
<form method="${usetype}" action="/cgi-bin/testdoc/alltogether_demo.sh">
Name: <input name="username" type="textarea" size="30" width="30" maxlength="30" value="$
{name}">
Age: <input name="age" type="textarea" size="5" width="5" maxlength="5" value="${age}">
<input type=submit value="Submit Data"><input type=reset>
</form>
EOF
} # end display_form

# **********************************************************
# - Depending on whether get or post method was used save the
#   data passed.
# - display the input form using the other action method
# - show the data provided
# - end the webpage
# **********************************************************
case "${REQUEST_METHOD}" in
  "POST") HTTPDATABUFFER=`cat`   # get the daya passed
          extract_fields         # so we can put values in next form display
          display_form "get"     # display the next input form
```

```
        ;;
  "GET")  HTTPDATABUFFER="${QUERY_STRING}"  # get the data passed
        extract_fields        # so we can put values in next form display
        display_form "post"    # display the next input form
        ;;
  *)      throw_unsupported_page "${REQUEST_METHOD}"
        ;;
esac
show_entered_data
echo "</body></html>"
exit 0
```

7. Additional Information to take you even further

You will have seen in the cookie section that cookies are set by providing header values. There are a lot of available headers that you can use such as setting http codes (redirect, cache control, spoofing X-Forwarded-* information etc).

Information on all available headers is easily found on Wikipedia at

https://en.wikipedia.org/wiki/List_of_HTTP_header_fields

So you have plenty to play with going forward.