A New Blade for the Old Swiss Army Knife

Effective Shell Scripting for the Web

John DiMarco

Techknowfile 2012

Shell Scripting

- Long-standing tool for system administration
- Ad-hoc data analysis
- Automation

- But WWW?

Programming for the Web

- 1990s: Perl CGIs
- 2000s: PHP, ASP.NET, Java servlets
- 2010s: Python, Ruby, etc.

- What about shell scripting?

helloworld.cgi

```
#!/bin/sh
echo “content-type: text”
echo “”
echo “Hello, World!”
```

```
#!/bin/sh
echo “content-type: text/html”
echo “”
echo ‘<html><body>’
echo “Hello, World!”
echo ‘</body></html>’
```

Why no love for /bin/sh cgis?

- It’s simple, easy!
- Lots of people know how to write shell scripts.

- Then why are shell script cgi’s so rare?

The Pesky Parameter Problem

URL Encoding

- Parameters are passed via the URL

http://my.site.edu/myscript.cgi?param1=something&param2=somethingelse

MIME Encoding

- Parameters are passed as “mime-encoded” data

```
-----------------------------23281168279961
Content-Disposition: form-data; name=“who"

Jane Doe
-----------------------------23281168279961
Content-Disposition: form-data; name=“dat"

datadatadatadatadata
-----------------------------23281168279961--
```

Parameters are Pervasive

- All interesting web code needs parameters
- Especially forms!
  - Post the HTML form
  - Parse the submitted form fields as parameters
- Perl, PHP, Python, etc. all have built-in libraries for this
- What about shell scripts?

Shell Script Parameter Parsing

```
...
_F_VAL="$_F_VAL""++"
    _F_TMP=

    while [ "$_F_VAL" != "" -a "$_F_VAL" != "+" -a
"$_F_VAL" != "++" ] ; do
        _F_TMP="$_F_TMP""`echo $_F_VAL |
cut -d + -f 1`"
        _F_VAL=`echo $_F_VAL | cut -s -d + -f 2-`

        if [ "$_F_VAL" != "" -a "$_F_VAL" != "+" ]
; then
            _F_TMP="$_F_TMP"" "
        fi
    done

    if [ ${DEBUG:-0} -eq 1 ] ; then
        echo "  " vrs=$_F_TMP 1>&2
    fi
    _F_TMP="$_F_TMP""%%"
    _F_VAL=

    while [ "$_F_TMP" != "" -a "$_F_TMP" != "%"
] ; do
        _F_VAL="$_F_VAL""`echo $_F_TMP |
cut -d % -f 1`"
        _F_TMP=`echo $_F_TMP | cut -s -d % -f
2-`

        if [ "$_F_TMP" != "" -a "$_F_TMP" !=
"%" ] ; then
            if [ ${DEBUG:-0} -eq 1 ] ; then
                echo "  " got hex "%" $_F_TMP
1>&2
            fi
            _F_HEX=`echo $_F_TMP | cut -c 1-2
| tr "abcdef" "ABCDEF"`
            _F_TMP=`echo $_F_TMP | cut -c 3-`
_F_VAL="$_F_VAL""`/bin/echo \"0\"echo
"16i8o"$_F_HEX"p" | dc\`"
        fi
    done
....
```

Why?

- Hard to get right
  - %-style representations for many characters (e.g. %20 for space)
  - Different delimiters than /bin/sh (eg. &)
  - Quoting challenges
- Hard to make secure
  - IFS
- Slow
- Shell Scripts are supposed to be ***short***.

What to do?

- Consensus view:
  - Don't use shell script cgi's at all.
- My view:
  - "Consensus" doesn't really understand shell scripting.

Shell scripting philosophy

- Shell offers "programming glue": most of the real work is done by commands.
- Commands should do "one thing well".
- Commands follow standard /bin/sh conventions to work together.
- Commands provide results that can be easily be used by other shell commands.
  - Pipe-able data, files

I wrote: **urldecode**

- Command that parses url-encoded and mime-encoded data.
- Does "one thing well": www parameters
- Uses standard shell conventions
- Uses stdin, stdout, stderr
- Converts parameters to files

- Written in Lex and C (not sh!)

Why **urldecode**?

- Shell scripts can't easily handle MIME-encoded and URL-encoded data.
- Shell scripts can easily handle a bunch of text files in a directory.
- **urldecode** converts one to the other.

urldecode usage

- Input is url-encoded or mime-encoded data
  - stdin
- Specify "parameters of interest" as arguments
  - Simple regular expressions (wildcards) supported.
- Parameter contents, if present, written to files in a parameter directory.
  - Name of file is the name of the parameter
  - Contents of file is the value of the parameter
  - Parameter directory must not exist
- Parameter name is output to stdout when seen
  - For use via pipes to downstream commands, or for debugging

urldecode usage con't

- URL-encoded data (e.g. in $QUERY_STRING)

```
echo "$QUERY_STRING" | urldecode -d $TMPDIR parm1 parm2 |
while read p; do
    case "$p" in
        "parm1") # handle $TMPDIR/parm1…
            ;;
        "parm2") # handle $TMPDIR/parm2
            ;;
    esac
done
```

- MIME-encoded data via stdin

```
urldecode -D $TMPDIR parm1 parm2 | while read p; do …
```

Web CGI variables refresher

- $REQUEST_METHOD
  - "GET" if this is a display of a web page
    - Parameter data if any supplied via $QUERY_STRING
    - URL-Encoded data only
  - "POST" if this is a web form being returned
    - Parameter data is supplied on stdin
- $QUERY_STRING
  - URL-encoded data from URL, if any (GET)
  - E.g. "parm1&parm2&parm3"
- $UNIQUE_ID
  - An identifier that is unique to this run of the cgi

Simple Example: PS2PDF.cgi

- Converts postscript input to PDF output

```
#!/bin/sh
TMPDIR="/tmp/$$$UNIQUE_ID"
exec 2>/dev/null # ignore errors (can redirect for debugging)
case "$REQUEST_METHOD" in
        "GET") # Display web form
        …
        ;;
        "POST") # Convert to PDF
        …
        ;;
esac
```

PS2PDF: display web form

```
…
"GET") # display a web page asking for a PS file as input.
        echo Content-type: text; echo
        echo '<html><body>'
        echo '<form action=PS2PDF.cgi enctype="multipart/form-
data" method=post>' echo 'PS file:<input type=file name=psfile>'
        echo '<input type=submit value=send>'
        echo '</form></body></html>'
;;
…
```

PS2PDF: convert to PDF

```
"POST") # Convert the specified PS file to PDF and output it.
        trap "rm -rf $TMPDIR" 0 1 2 15
        urldecode -D $TMPDIR psfile >/dev/null
        if ps2pdf $TMPDIR/psfile $TMPDIR/pdffile; then
                echo Content-type: application/pdf; echo
                cat $TMPDIR/pdffile
        else
                echo Content-type: text; echo
                echo Conversion failed.
        fi
 ;;
…
```

What about variable number of parameters?

- What if line-1 through line-N?
- **Urldecode:** wildcarded parameter names
  - Use sh wildcards or POSIX regular expressions

  e.g. *urlecode –d $TMPDIR 'line-[1-9][0-9]*'*

What about filename, content-type?

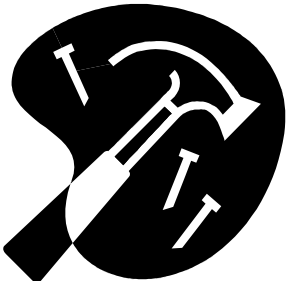
- MIME-encoding: filename, content type?
  - **urldecode** ignores it when creating the file
    - Filename is the argument name, not the filename specified in the MIME header, if any
    - Contents are the (binary) contents, without any regard for content-type
  - argname filename="myfilename" and contenttype="content/type" is reported on stdout
    - Script can parse this, if interested.

How about user authentication?

- Basic or Digest Authentication via https
  - SSL (https) takes care of snoopers
- Basic Authentication: passwd encryption same as UNIX /bin/passwd
- User name passed to script as $REMOTE_USER

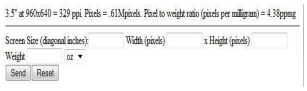
```
<Directory "/my/dir“>
  Options ExecCGI
  AddHandler cgi-script .cgi
  AuthType Basic
  AuthName “Users“
  AuthUserFile “/passwd“
  Require valid-user
</Directory>
```

So what can you build with all this?



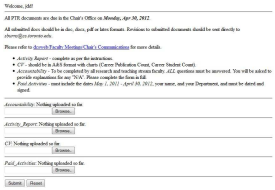
Screen/Weight Density Calculator

- screenspec.cgi: 47 lines (+ inline documentation)
- Input screen size, resolution, and device weight (g, kg, lb, or oz)
- Compute pixel density and weight ratio of portable device
  - Pixels per inch
  - Pixels per gram



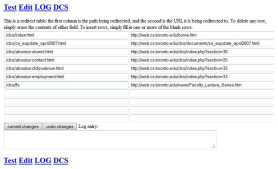
Document Uploader

- ptr.cgi: 48 lines
- Uploader for faculty documents for PTR.
- Full authentication
- Configurable document names, destination, message
- Standardized document naming for easy processing.



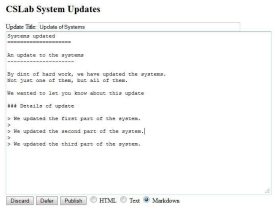
Redirects Maintenance

- Redirects.cgi: 127 lines
  - Maintain web server redirects via web page
  - Version control, revert to previous version, revision logging, differences between versions
  - Full user access control



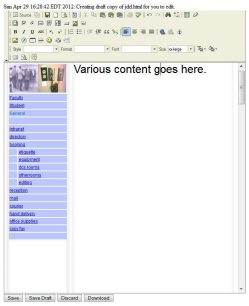
Blog update editor

- update.cgi: 153 lines
- Create blog entries
  - Blosxom format
  - HTML, text or markdown
- Preview, defer, publish, discard
- Same script supports multiple blogs
  - Unique config file/blog



In-line Web File Editor

- Edit.cgi: 150 lines (+ inline documentation)
  - Edit html files in your web browser
  - Uses javascript WYSIWYG editor: either *TinyMCE* or *FCKeditor* supported
  - Per-file user authentication/control.
  - Concurrency control, versioning & edit logs (RCS)
  - Draft management



When to use shell cgis?

- Use when
  - Need something quick, easy
  - Want to leverage shell scripting expertise
  - Want to leverage UNIX-style commands
- Not designed for:
  - Complex/large web applications
- Get urldecode from www.cs.toronto.edu/~jdd