

# Using Web Forms to access UniVerse

Ross Morrissey  
OSDA Winter 2002 Newsletter

The last article (UniVerse Intranet Access) took advantage of a primitive HTML concept, the ISINDEX query. This allowed very simple command line arguments to be passed to a script and allowed us to run simple one-line commands. There are two big limitations with ISINDEX – we have little control over the user interface, and we can't pass anything more than rudimentary strings. We can overcome both of these limitations by switching to a web form.

After a brief introduction to forms, we'll look at how a couple of popular web sites use web forms and speculate about how UniVerse would interact with similar forms, both extracting the data and building the forms themselves. Again, we will assume that we're running UniVerse on a Unix system with an Apache web server (although most of this will apply to other configurations). You'll need a little knowledge of Unix, and I'm going to gloss over a couple of complications that I'll touch on at the end. They shouldn't impact most applications in an Intranet environment.

## Quick Forms Background

Everybody who's visited a website and done anything more than follow a link has used a web form. There are two components to web forms – the form tags interpreted by the web browser, and the CGI (common gateway interface) scripts that handle the data on the web server end. Based on the **input** tags within the form, the user enters text, selects options, and requests that the browser submit their encoded data to the web server for a specified **action**. The server passes the user request to the specified script, which decodes the request and generates a response for the user.

## Capturing Data Using Forms – google.com

Here is an actual example of a chunk HTML from <http://www.google.com>:

*A useful attribute of the internet is the ability to eavesdrop on the HTML source code in use. Find an example you'd like to emulate, and go to View – Source in your web browser. You'll be amazed how much you learn. In this example, we're focusing on the forms aspect, but note the way google.com use the table and &nbsp; tags to cleanly organize their page's appearance – in any browser.*

```
<form action="/search" name=f>  
<table cellspacing=0 cellpadding=0>  
<tr>  
<td width=75>&nbsp;</td>  
<td align=center>  
<input maxLength=256 size=55 name=q value="">  
<br>
```



```
<input type=submit value="Google Search" name=btnG>  
http://www.google.com/search?q=osda&btnG=Google+Search
```

### On to the back end...

Now let's assume that the smart people at google.com are using UniVerse to handle their web queries. What might their program look like?

They'd need an "action" script "search" in a directory that was executable by their web server. This script would need to pass the arguments to UniVerse somehow, and in this case, we'll assume that it does so by invoking UniVerse directly. The web server places the arguments in a Unix shell variable QUERY\_STRING that we can access from UniVerse.

The Unix Script **search** (with read and execute permissions for everybody):

```
#!/bin/sh  
echo Content-type: text/html  
echo  
/usr/opt/uv/bin/uv GET.SEARCH
```

*This script **echos** a couple of lines to tell the web server that we're going to output text to be returned to the browser. By embedding this in the shell script we can ensure that any error messages produced in UniVerse are passed back to the browser to make debugging easier.*

In the same directory, we pop into UniVerse and look at the cataloged program **GET.SEARCH**:

```
EXECUTE "SH -c 'echo $QUERY_STRING'" CAPTURING PARAMETERS  
CONVERT "&=+" TO "@AM:@VM." " IN PARAMETERS  
NUMBER.OF.PAIRS = DCOUNT(PARAMETERS,@AM)  
NAME = ""; VALUE = ""  
FOR I = 1 TO NUMBER.OF.PAIRS  
  NAME<I> = PARAMETERS<I,1>  
  VALUE<I> = PARAMETERS<I,2>  
NEXT I  
  
LOCATE "q" IN NAME SETTING POS THEN QUERY = VALUE<POS> ELSE QUERY = ""  
  
LOCATE "btnG" in NAME SETTING POS  
  THEN CALL GOOGLE.SEARCH(QUERY,RESULT)  
  ELSE CALL LUCKY.SEARCH(QUERY,RESULT)  
  
PRINT RESULT  
  
RETURN
```

When we run this, we'll break the QUERY\_STRING into clean name and value pairs:

<b>Name</b>	q	btnG
<b>Value</b>	osda	Google Search

A routine that does the real work, GOOGLE.SEARCH is invoked, and the result is printed.

**Now let's look at some of the more interesting parts of this program.**

**EXECUTE "SH -c 'echo \$QUERY\_STRING'" CAPTURING PARAMETERS**

This retrieves the portion of the URL after the ? that Apache has placed in the QUERY\_STRING Unix environment variable:

<http://www.google.com/search?q=osda&btnG=Google+Search>

*There are many other environment variables that are set by the web server for each of these requests, two that you might find useful are REMOTE\_HOST or REMOTE\_ADDR that store the hostname and IP address of the requesting client. You would make use of the CONTENT\_LENGTH variable if you used the POST method.*

**CONVERT "&=+" TO @AM:@VM:" " IN PARAMETERS**

We're rolling three translations into one here. Each of the name/value pairs in QUERY\_STRING is separated by an &; by converting these to attribute marks, we get an easy way to manage the pairs. Each pair is in the format name=value; by converting each = to a value mark we can pull these out of each pair easily. The + sign is used to represent a space (as in Google+Search).

*Another good reason to swap these out in one step up front is that these three special characters may occur URL encoded within values. This allows us to enter strings like "&UFD&" or "C++" in a text box. We'll look at URL encoding later.*

**LOCATE "q" IN NAME SETTING POS THEN QUERY = VALUE<POS> ELSE QUERY = ""**

Pull the user's query out of the appropriate value. You will have a statement like this for every user-populated input field in the form.

**LOCATE "btnG" in NAME SETTING POS  
THEN CALL GOOGLE.SEARCH(QUERY,RESULT)  
ELSE CALL LUCKY.SEARCH(QUERY,RESULT)**

Here we use the name only to choose a subroutine to execute. The name and value in this case are fixed – if the name is "btnG" then the value is always "Google Search". They had to select of the buttons for us to see the script, so if it wasn't the "Google Search" button, they must have selected "I'm Feeling Lucky"

*Always use the name as a trigger for program logic. The user could have entered "Google Search" as their query and if we had searched on value to trigger the routine, we wouldn't know which button the user actually pressed. You have control over all the names, but not all the values.*

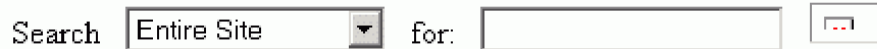
## **Form generation at walmart.com**

We use forms to create dynamic content for our users. Why not create dynamic forms too? This is a small step in the direction of content management, where we handle content and presentation separately. There are two advantages to handling forms this way: the user experience is enhanced, and maintenance is much cleaner – the options within the form are table driven.

Let's look at another real world example. Here is a portion of the page we are redirected to automatically when we go to the <http://www.walmart.com> home page:

```
<html><head><title>Walmart Fragment</title></head><body>
<form action="/catalog/search.gsp" method="GET" target="_top"> </td></tr>
<tr>
<td class=text1 bold nowrap>Search</td>
<td width=7 nowrap>&nbsp;</td>
<td class=text2>
<select name="search_constraint" >
<option value="0">Entire Site
<option value="3944">Electronics
<option value="5426">Photo
<option value="4104">Music
<option value="4096">Movies
<option value="3920">Books
<option value="4171">Toys
<option value="4125">Sporting Goods
<option value="3891">Jewelry
<option value="4044">Home & Garden
<option value="2637">Gifts
<option value="2636">Video Games
</select>
</td>
<td width=5 nowrap>&nbsp;</td>
<td class=text1 bold>for:</td>
<td width=7 nowrap>&nbsp;</td>
<td>
<input type="text" size="20" maxlength="100" value="" name="search_query" class="text2"></td>
<td width=5 nowrap>&nbsp;</td>
<td>
<input type=image src="/i/if/cat/btn/bp_find_btn.gif" name="Continue" alt="Find" width=36 height=20
value="Find" border=0></td></tr>
<tr>
<td colspan=9>
</form>
</body></html>
```

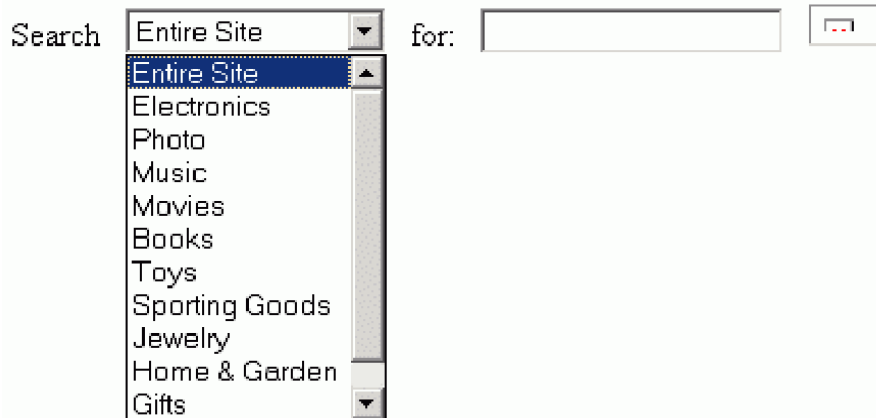
Here is what this fragment looks like on its own:



Search  for:

*Note that the graphic on the right is not available in our little sample – it's local to walmart.com.*

If I click on the down-arrow, I get a list of options:



Search  for:

- Entire Site
- Electronics
- Photo
- Music
- Movies
- Books
- Toys
- Sporting Goods
- Jewelry
- Home & Garden
- Gifts

Let's imagine that the Walmart team are using UniVerse to dynamically generate their forms. What might their scripts look like?

To start with, they use the same Unix script, with a different name and target program, **search.gsp**:

```
#!/bin/sh
echo Content-type: text/html
echo
/usr/opt/uv/bin/uv WAL.SEARCH
```

*Why don't we use a different script, perhaps a "get-ready-for-search" script? Actually, using the same script allows us to handle the case where a user is just arriving at the site and the case where they submit an empty form the same way.*

In the same directory, we pop into UniVerse and look at the somewhat familiar looking cataloged program **WAL.SEARCH**:

```
EXECUTE "SH -c 'echo $QUERY_STRING'" CAPTURING PARAMETERS
CONVERT "&=+" TO "@AM:@VM:" " IN PARAMETERS
NUMBER.OF.PAIRS = DCOUNT(PARAMETERS,@AM)
NAME = ""; VALUE = ""
```



## FILE.HTML.TEMPLATES WAL.SEARCH.SUFFIX

```
</select>
</td>
<td width=5 nowrap>&nbsp;</td>
<td class=text1 bold>for:</td>
<td width=7 nowrap>&nbsp;</td>
<td>
<input type="text" size="20" maxlength="100" value="" name="search_query" class="text2"></td>
<td width=5 nowrap>&nbsp;</td>
<td>
<input type=image src="/i/if/cat/btn/bp_find_btn.gif" name="Continue" alt="Find" width=36 height=20
value="Find" border=0></td></tr>
<tr>
<td colspan=9>
</form>
</body></html>
```

CONSTRAINT 3944  
Electronics

CONSTRAINT 5426  
Photo

CONSTRAINT 4104  
Music

CONSTRAINT 4096  
Movies

CONSTRAINT 3920  
Books

CONSTRAINT 4171  
Toys

CONSTRAINT 4125  
Sporting Goods

CONSTRAINT 3891  
Jewelry

CONSTRAINT 4044  
Home & Garden

CONSTRAINT 2637  
Gifts

CONSTRAINT 2636  
Video Games

When the script is invoked directly by the web server and PARAMETERS aren't supplied, we build a web page to display to the user. First, we retrieve and display the

fixed top portion of the page, then cycle through CONSTRAINT choices for a drop-down selection, then display the fixed remainder of the page. If data was supplied, we hand it off to another program to do the work. Notice that this program really isn't very long, and more than a quarter of it deals with error reporting.

### Let's look at some of the more interesting parts of this program...

```
EXECUTE "SH -c 'echo $QUERY_STRING' CAPTURING PARAMETERS
CONVERT "&=+" TO "@AM:@VM." " IN PARAMETERS
NUMBER.OF.PAIRS = DCOUNT(PARAMETERS,@AM)
NAME = ""; VALUE = ""
FOR I = 1 TO NUMBER.OF.PAIRS
  NAME<I> = PARAMETERS<I,1>
  VALUE<I> = PARAMETERS<I,2>
NEXT I
```

The first block of eight lines is identical to the previous program. We'll move this into a subroutine soon.

```
IF NAME<1> EQ "" THEN ;* The user hasn't seen the page yet!
```

The action script is being called directly from the address line of the browser, not as the result of a form submission. This is the trick that allows us to use the same code to populate an initial form as we use to respond to an empty or invalid request.

```
OPEN "", "HTML.TEMPLATES" TO FILE.HTML.TEMPLATES ELSE
  PRINT "SYSTEM BUSY, TRY LATER <! Couldn't open HTML.TEMPLATES!>"
  RETURN
END
```

This is displayed as: SYSTEM BUSY, PLEASE TRY LATER

The user sees a familiar, easy to understand message, while hiding the helpful portion from their view. The portion of the string inside the <! !> delimiters is not displayed by the browser. The entire string is present in the source for debugging purposes.

*If you "hide" items in html comments, make sure they really are comments – aids to writing and debugging the html. Don't hide data from the user there. They will find it.*

```
PRINT "<option value=" : CONSTRAINT.ID : ">" : CONSTRAINT.NAME
```

This is where we transform:

```
CONSTRAINT 3944
Electronics
```

Into:

```
<option value="3944">Electronics
```

*Note that for this example, we're displaying the selection values in SELECT order; adding a sort key to get any desired ordering would be straight-forward.*

LOCATE "search\_constraint" IN NAME SETTING POS

This is analogous to "LOCATE "q" IN NAME" from the previous program. We know at this point that the user has submitted the form (and of course they may have left the text field empty).

### **Let's look at some of the more interesting pieces of data...**

FILE.HTML.TEMPLATES WAL.SEARCH.PREFIX

```
<html><head><title>Walmart Fragment</title></head><body>
<form action="/catalog/search.gsp" method="GET" target="_top"> </td></tr>
<tr>
<td class=text1 bold nowrap>Search</td>
<td width=7 nowrap>&nbsp;</td>
<td class=text2>
<select name="search_constraint" >
<option value="0">Entire Site
```

action="/catalog/search.gsp"

This is the script run on the server to extract the parameters and formulate a response.

<select name="search\_constraint" >

This is the drop-down box tag. The drop-down options follow.

<option value="0">Entire Site

This is the first option. "Entire Site" is displayed to the user, but "0" will be returned to the server if this option is chosen.

FILE.HTML.TEMPLATES WAL.SEARCH.SUFFIX

```
</select>
</td>
<td width=5 nowrap>&nbsp;</td>
<td class=text1 bold>for:</td>
<td width=7 nowrap>&nbsp;</td>
<td>
<input type="text" size="20" maxlength="100" value="" name="search_query" class="text2"></td>
<td width=5 nowrap>&nbsp;</td>
<td>
<input type=image src="/i/if/cat/btn/bp_find_btn.gif" name="Continue" alt="Find" width=36 height=20
value="Find" border=0></td></tr>
<tr>
<td colspan=9>
</form>
</body></html>
```

input type="image"

Instead of a "submit" type, we use "image" to supply our own button.

### Putting it together – a general script

Let's create a general script that can handle the interaction with forms from the UniVerse side given what we've learned from the pros. It should dispatch the name/value pairs to another UniVerse routine for specific handling.

Lets start with a unix script patterned after the last one. We'll call it **uvform**:

```
#!/bin/sh
echo Content-type: text/html
echo
/usr/opt/uv/bin/uv UVGET
```

Our UniVerse **UVGET** routine shares code from our previous efforts, and has a couple of new wrinkles:

```
DEFFUN UNESCAPE.URL(RAW.VALUE)
$OPTIONS -M

EXECUTE "SH -c 'echo $QUERY_STRING'" CAPTURING PARAMETERS
CONVERT "&+=" TO "@AM:@VM:" " IN PARAMETERS
NUMBER.OF.PAIRS = DCOUNT(PARAMETERS,@AM)
DIM VALUE(NUMBER.OF.PAIRS)
FOR I = 1 TO NUMBER.OF.PAIRS
  NAME<I> = PARAMETERS<I,1>
  RAW.VALUE = PARAMETERS<I,2>
  VALUE(I) = UNESCAPE.URL(PARAMETERS<I,2>)
NEXT I

LOCATE "uvroutine" IN NAME SETTING POS THEN
  UVRoutine = VALUE(POS)
  CALL @UVRoutine(NAME,MAT VALUE,RESULT)
  PRINT RESULT
END ELSE PRINT "SYSTEM BUSY <- Couldn't find uvroutine ->"
RETURN

FUNCTION UNESCAPE.URL(URL.STRING)

PERCENTS = COUNT(URL.STRING,"%")
IF PERCENTS THEN
  URL.STRING.LENGTH = LEN(URL.STRING)
  CLEAN.STRING = STR("~",URL.STRING.LENGTH - PERCENTS*2)
  X = 1
  Y = 1
  LOOP
  IF URL.STRING[X,1] EQ "%" THEN
```

```

    CLEAN.STRING[Y,1] = CHAR(OCONV(URL.STRING[X+1,2],"MCX"))
    X += 2
END ELSE
    CLEAN.STRING[Y,1] = URL.STRING[X,1]
END
WHILE (X LT URL.STRING.LENGTH) DO
    X += 1
    Y += 1
REPEAT
END ELSE
    CLEAN.STRING = URL.STRING
END

RETURN (CLEAN.STRING)

```

### Let's look at some of the more interesting parts of this program...

```
DEFFUN UNESCAPE.URL(RAW.VALUE)
```

When we pass arguments as part of the URL or web page address (as we do with the GET method of CGI forms), we must "URL encode" these arguments. This means taking any special characters and converting them to a string consisting of a % followed by the hexadecimal representation of the ascii character. The classic example is %20, a space.

*You'll see this frequently in web addresses on Microsoft platforms where the document referenced has a filename containing spaces.*

```
DIM VALUE(NUMBER.OF.PAIRS)
```

By using a dimensioned array to hold our values, we can include attribute and value marks in the data. We may not enter them in text forms, but other uv-aware clients may wish to use this method to pass data.

```
VALUE(I) = UNESCAPE.URL(PARAMETERS<I,2>)
```

We pass the value string to a routine that does the conversion from URL encoded string to "unescaped" string.

```
LOCATE "uvroutine" IN NAME SETTING POS THEN
    UVROUTINE = VALUE(POS)
    CALL @UVROUTINE(NAME,MAT VALUE,RESULT)
    PRINT RESULT

```

This is where the form is dispatched to the UniVerse routine that does the heavy lifting. The uvroutine is specified in the form with a "hidden tag" – for example:

```
<input name="uvroutine" value="FILE.COUNT" hidden>
```

This tag is not displayed to the user, but is passed in with the name and value tags. This subroutine name is coded in the html that triggered the original query OR directly in the URL used to initially populate the web page.

*What will happen if we CALL @UVROUTINE and the subroutine referenced by UVROUTINE is not cataloged? We'll generate an error message. The error message will be passed directly back to the web browser. This is the advantage of echoing the first two lines of the html response in the shell script. If we rely on UniVerse to echo these initial lines, we can't pass UniVerse errors to the browser, the users session appears to hang, and we get an unhelpful "premature end of script headers" error in the Apache error log.*

FUNCTION UNESCAPE.URL(URL.STRING)

The logic from this routine is a BASIC version of **unescape\_url**, the well-known C routine for URL decoding.

### **An example application**

The user is supplied this URL:

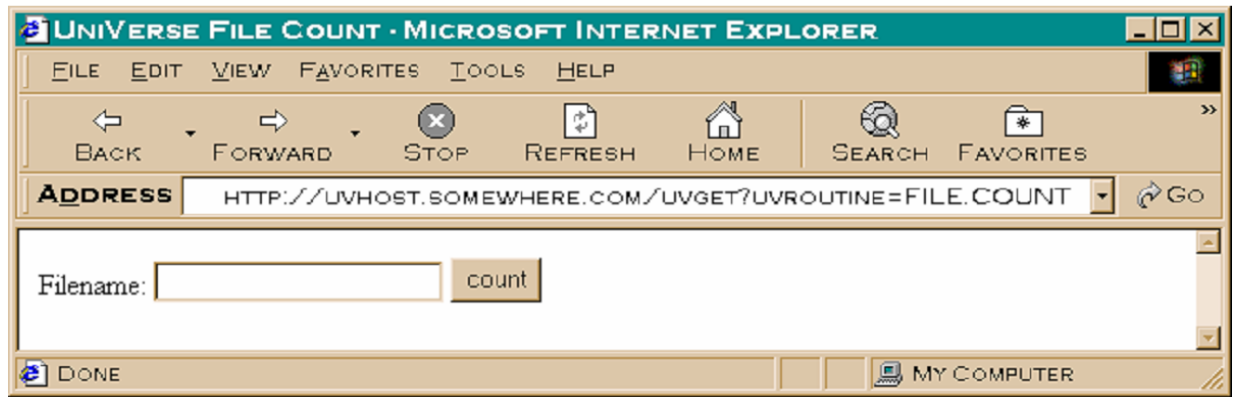
<http://uvhost.somewhere.com/uvget?uvroutine=FILE.COUNT>

When they click on the URL, the server invokes the uvget script, which invokes UV.GET which parses out the uvroutine value of FILE.COUNT. The UniVerse routine FILE.COUNT is called with only one value set:

```
SUBROUTINE FILE.COUNT(NAME,MAT VALUE,RESULT)

LOCATE "file" IN NAME SETTING POS THEN
  EXECUTE "COUNT " : FILE CAPTURING RESULT
END ELSE
  RESULT = '<html>'
  RESULT := '<head><title>UniVerse File Count</title></head>'
  RESULT := '<body>'
  RESULT := '<form action="uvget">'
  RESULT := '<input type=hidden name=uvroutine value="FILE.COUNT">'
  RESULT := 'Filename:'
  RESULT := '<input name=file>'
  RESULT := '<input type=submit name=button value="count">'
  RESULT := '</form>'
  RESULT := '</body>'
  RESULT := '</html>'
END
RETURN
```

The first time through this routine, the name file is not found, and the RESULT returned to be printed on the browser is the initial web page for display:



When a filename is supplied and the count button is pressed, the value of file, button, and the hidden value FILE.COUNT are passed to the script. In FILE.COUNT, file has a value and EXECUTE "COUNT " : FILE CAPTURING RESULT Is executed.

The result, whether it is "40 records counted" or "blart not found in VOC" is sent back to the browser. It is trivial to wrap the result in special formatting, or even reproduce the form on the results page. One of the benefits is that the user can bookmark the results for future reference – a real help when several fields have been filled in.

We can now perform a COUNT &UFD& - something useful you can't do with the ISINDEX function.

### Limitations

This GET method of calling CGI scripts has limitations with the amount of data that can be passed because of its "command line" nature. There is a POST method where the parameters are passed to the script via Unix standard input. Traditionally the GET method is used for queries, and the POST method is used for updating data.

Executing uv directly from within the shell script can sometimes bump up against "printer segment removed" messages where a defunct uv session was associated with the pid. This will echo a message back to your browser, but is otherwise harmless. You may experience problems with locks too. There are a couple of methods of getting around this that are beyond the scope of this article.

Next: a couple of UV-centric web applications