Server-Agnostic Perl/DCL
CGI Programming with WASD and OSU

Dick Munroe
Cottage Software Works, Inc

Overview

In January of 2003, a client of mine decided to switch from Purveyor, on which they were running their secure commerce web site, to the WASD web server. The OSU server was also in use at the site, but my clients decided that the features provided by WASD were better than those available in OSU. I was asked to install the WASD server and move the secure portion of their applications from Purveyor to WASD. During this job I suggested that my clients stop using the OSU server as well, purely for support reasons. After all, one web server is less work to support than two. To avoid a "flag day", where we would have to switch an entire application from one web server to another, I designed a general framework that allowed them to use their current DCL/Basic CGIs in either environment. Since the clients followed a consistent pattern for implementing their CGIs, it was also possible to write tools that prepared their applications for execution under either WASD or OSU. In turn, this made the switch from the Purveyor/OSU environment to the WASD/OSU environment quick and simple, while leaving the elimination of OSU a decision that could be made on a case-by-case basis.

I was so impressed with the quality of the WASD server[1], its documentation, the provided debugging tools, and the commitment of the WASD development group that I decided to switch from OSU to WASD at my site.

Hedging My Bets

Having made the decision to switch from OSU to WASD, I wanted to hedge my bets. A fair portion of my site has functionality implemented by CGIs. I didn't want to lock myself irretrievably into WASD (or into any specific server if possible). So I had to look into the same issues that my clients had, i.e., how to build CGIs that run under all servers.

While I do occasionally write CGIs in a compiled language like C, the job can generally be done quite easily with DCL, Perl, or some combination of both. The net result is that most of the CGIs at my site are written in one of these languages.

So, what execution environments does each server offer for DCL and Perl?

- OSU

  DCL can execute only under the "script server," which uses DECnet to create an execution environment and communicate with the CGI.

  Perl also executes through the "script server" environment with some special case code in WWWEXEC.COM to ensure that Perl CGIs have an appropriate execution environment. OSU also provides an execution environment using the FastCGI interface specification.

[1] I believe the WASD and OSU servers to be, more or less, equivalent in terms of performance and overall capabilities. The WASD web server and environment is much better documented than the OSU distribution and is therefore much easier to use and manage, giving WASD the edge.

- WASD

  DCL can execute in one of three environments. The first environment is an OSU emulation. In practice I saw no differences between the OSU server and the WASD OSU emulation. The second environment involves execution of the CGI as a sub-process of the server. The third, CGIplus (analogous to FastCGI), dedicates a process to running a DCL CGI.

  As with DCL, Perl can also execute under the OSU emulation environment, directly as a sub-process, or with a dedicated process. WASD provides one additional execution environment, PerlRTE. PerlRTE is a persistent Perl interpreter, analogous to mod_perl. CGIplus is a persistent Perl "server", analogous to FastCGI: a process dedicated to the repetitive execution of a single Perl script.

PerlRTE and CGIplus are performance optimizations. They address the two principal overhead components in execution of any Perl CGI or program:

1. Creation of the process running Perl and loading of the Perl interpreter (PerlRTE)
2. Loading of the Perl modules necessary for execution of the CGI or program (CGIplus)

PerlRTE creates a process and loads the interpreter but can execute any Perl.CGI. The cost of creating the process and loading the Perl interpreter is amortized across **all** CGIs using PerlRTE.

CGIplus creates a process and loads the Perl interpreter (when executing a Perl CGI) but then goes on to load and execute the CGI. The Perl CGI is wrapped in a loop and is available for "instant" execution the next time that CGI is invoked. The cost of creating the process, loading the Perl interpreter, and loading all the necessary Perl modules required by the CGI is amortized across the number of invocations of a **specific** CGI. Each CGIplus-enabled CGI requires a dedicated process for execution. Unfortunately, each of these environments differs in small ways that must be accounted for in a server-agnostic CGI.

Perl CGI Programming in the Different Environments

This document is **not** a tutorial on how to program CGIs or how to program in Perl. Both of those topics have been covered in the literature more thoroughly and far better than I can do. However, a quick discussion of the basics is in order.

Outside of mod_perl, the most commonly used CGI programming interface is the venerable CGI.pm. It provides access to the CGI environment variables, access to query, form, and multipart form data. CGI.pm can also generate http protocol headers and many of the standard HTML tags, making creation of dynamic content much easier.

Listing 1 shows the basic structure of a simple Perl CGI using CGI.pm.

```
use strict;
use 5.6.1 ;
use CGI ;
my $theCGI = new CGI ;
print $theCGI->header('text/plain') ;
my @theParameterNames = $theCGI->param() ;
print "The Parameter/Value pairs:\n" ;
print "--------------------------\n\n" ;
foreach (sort @theParameterNames)
{
    my @theValue = $theCGI->param($_) ;
    if ($#theValue)
    {
        for (my $i = 0; $i <= $#theValue; $i++)
```

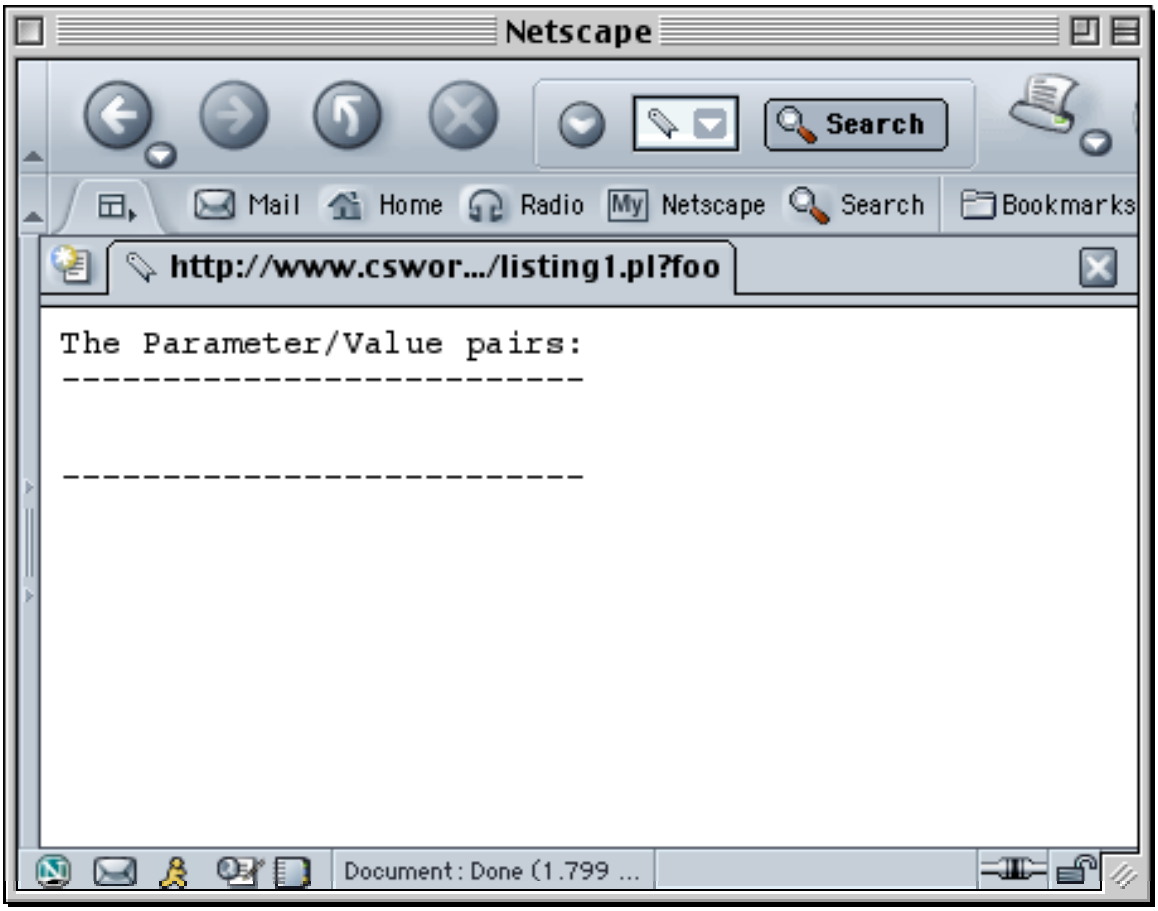
```
        {
            print $_,"[$i] = ",$theValue[$i],"\n" ;
        }
    }
    else
    {
        print $_," = ",$theValue[0],"\n" ;
    }
}
print "\n--------------------------\n" ;
```

Listing 1
CGI to list the names/values of query/form parameters.

As can be seen, the CGI is pretty straightforward. The CGI is invoked, goes through the hash of parameter names in alphabetical order and prints the values.

You can see the CGI in action at http://www.csworks.com/cgi-bin/vtj/listing1.pl?foo=1. If you download the source for this article and put the listing1.pl CGI in CGI-BIN:[000000] so your WASD server can find it, fire up a browser and try http://your.server.name/cgi-bin/listing1.pl?foo=1 you will, likely, see something like Figure 1. What happened to the value of foo? Why didn't it print?

Figure 1
What happened to the Parameters?



If you didn't get a display at all or got an error message, you probably have a simple configuration issue. You need to tell WASD how to execute Perl code. Full details are available in the WASD documentation, but you need to make sure that there is a mapping from the .pl suffix to a bit of DCL that executes the Perl procedure in the right environment. The mapping is kept in HT_AUTH:HTTPD$CONFIG.CONF and at my site looks like this:

```
[DclScriptRunTime]
.pl @cgi-bin:[000000]perl.com
```

CGI-BIN:[000000]PERL.COM is:

```
$ define /user perl_env_tables clisym_global,lnm$process
$ perl "''p1'"
$ exit
```

and is provided with the WASD distribution in the right place for use. Once the linkage between file type (.pl) and execution environment (CGI-BIN:[000000]PERL.COM) is properly set up, you can move on.

The second possibility is the first lesson in server agnosticism. Both WASD and OSU prefix their CGI environment variables (those defined by the CGI Interface Specification) with "WWW_". This is intended to do two things:

1. Identify the CGI variables as belonging to the World Wide Web environment and
2. Prevent the standard CGI variable names from "masking" the equivalent DCL symbols or OpenVMS logical names, both of which are provided to Perl programs as part of the general environment available via the %ENV hash.

Unfortunately, we are dealing with Perl and CGI.pm whose roots are deep in the Unix world. The Unix CGI environment doesn't prefix its CGI environment variables with "WWW_". Rather the CGI environment variables are provided unmodified as per the CGI Interface Specification. Since WASD is prefixing the CGI environment variables with "WWW_", CGI.pm and any other Perl code following the CGI Interface Specification won't see the CGI environment variables.

In order to bolt Perl CGIs using CGI.pm to WASD you must add to your mapping files something like the following:

```
set /cgi-bin/vtj/*            cgiprefix=
```

before the mapping of the CGI via the exec occurs. Once these lines have been added, WASD will apply the specified prefix (none at all) to all CGIs in the cgi-bin/vtj directory. Of course you can make this as specific as you want, e.g.:

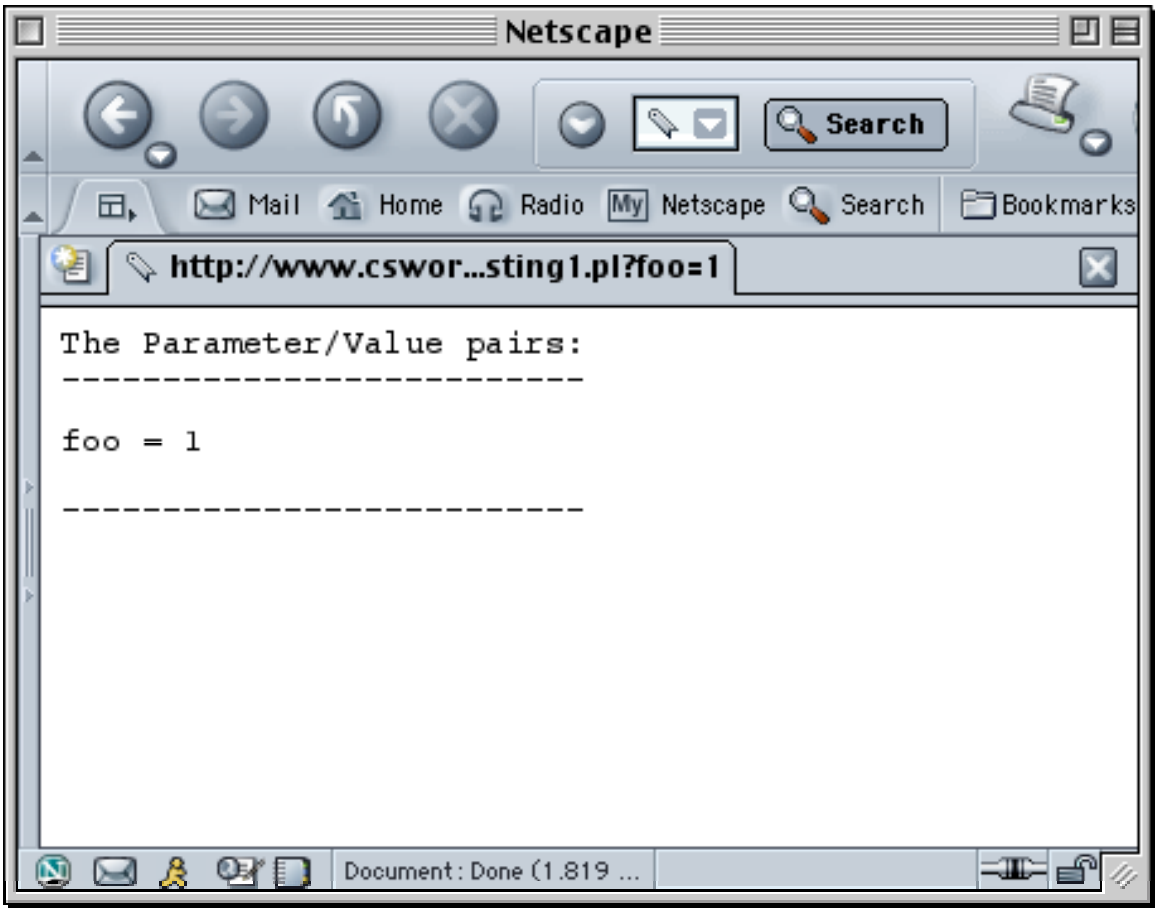
```
set /cgi-bin/vtj/listing1.pl        cgiprefix=
```

provides an empty (zero length) CGI prefix for only listing1.pl.

Once you've done this, reloaded the WASD server's mappings[2], and executed listing1.pl again on your server, you should see something like Figure 2.

[3] For the WASD server, HTTPD/DO=MAP reloads the mapping files, HTTPD/DO=RESTART restarts the server reloading the entire configuration. Both have the same effect on mappings, but the RESTART may be visible by your users.

Figure 2
Parameters with cgiprefix set appropriately.



The OSU server has functionally equivalent magic to avoid breaking CGI.pm and other U*x oriented CGI code. It's built into WWWEXEC.COM and is not configurable short of modifying WWWEXEC.COM. However, WWWEXEC.COM makes reasonable assumptions about the Perl CGI execution environment.

If you run listing1.pl under OSU or under WASD's OSU emulation, you'll find that the same code runs identically under OSU, WASD with OSU emulation and WASD sub-processes.

The next CGI execution environment provided by WASD is CGIplus. The WASD kit includes a Perl module, CGIplus.pm, which provides CGIplus support. CGIplus.pm provides a number of useful interfaces, including a test to see if CGIplus mode is active, a usage counter, writing to the standard output stream in binary mode, access to CGI environment variables, etc. Unfortunately, the interfaces are provided without formal documentation[3]. You have to figure out what's there by inspection of the various examples provided in the WASD distribution or the CGIplus.pm code itself.

[3] One of the things that attracted me to the WASD web server was the thoroughness and high quality of the user documentation produced by the author. Finding a spot where the documentation was less than thorough was quite a shock.

The basic mechanisms for using CGIplus are simple. First, wrap what used to be your CGI in a subroutine, load CGIplus[4], and call the subroutine using the CGIplus process manager, CGIplus::process. Listing 2 shows the "obvious" port to use CGIplus.

```
unshift @INC,"HT_ROOT:[SRC.PERL]" ;

use strict;
use 5.6.1 ;

use CGI ;
require CGIplus ;

CGIplus::process(\&doit) ;

sub doit
{
    my $theCGI = new CGI ;
    print $theCGI->header('text/plain') ;
    my @theParameterNames = $theCGI->param() ;
    print "The Parameter/Value pairs (Usage: ",
        CGIplus::usageCount(),"):\n"  ;
    print "-------------------------\n\n" ;
    foreach (sort @theParameterNames)
    {
        my @theValue = $theCGI->param($_) ;
        if ($#theValue)
        {
            for (my $i = 0; $i <= $#theValue; $i++)
            {
                print $_,"[$i] = ",$theValue[$i],"\n" ;
            }
        }
        else
        {
            print $_," = ",$theValue[0],"\n" ;
        }
    }

    print "\n-------------------------\n" ;
    if ($ENV{'QUERY_STRING') eq "eoj")
    {
        exit ;
    }
}
```

Listing 2
Port to CGIplus

If you run http://www.csworks.com/cgiplus-bin/vtj/listing2.pl?foo=1 repeatedly, changing the value of the foo query parameter, you'll notice two things. First, the format of the output changes abruptly. All of a sudden there are a few extra new lines in the output. Second, and much more

[4] Since CGIplus isn't part of the standard Perl distribution, the unshift/require pair is necessary to get CGIplus loaded and ready for use. You could copy CGIplus.pm into your Perl library tree (at my site, I would put it in PERL_ROOT:[LIB.VMS_AXP.5_6_1]) and then use either a "require" or "use" statement.

important, the value of foo doesn't change from invocation to invocation, although the displayed usage count tells you that the CGI is, indeed, getting invoked. What's wrong?

This is the second lesson in server agnosticism. It's related to the first insofar as the problem is with CGI.pm, but the causes are completely different. CGI.pm predates CGIplus.pm by quite some time and CGI.pm has never been modified to take the WASD specific CGIplus environment into consideration. CGI.pm **has** been modified to look for Active States PerlEx and CGIplus.pm attempts to take advantage of that fact by asserting that CGI.pm is running in a PerlEx environment. By asserting "PerlEx" mode CGIplus.pm causes CGI.pm to reset its internal persistent state every time a new CGI object is created. All well and good, but listing2.pl still doesn't work!

What emerges is an ordering problem. CGIplus.pm asserts "PerlEx" mode when CGIplus::process is **run**. CGI.pm checks for the PerlEx environment when it **loads** (at the time the require or use statement loading CGI.pm is executed). As written, the CGI.pm loaded in listing 2 believes that it's running in a standard CGI environment and it never finds out about the persistent CGIplus environment.

So the lesson is to make sure that your CGI interface library stays sane across all environments. It may be necessary to build the occasional shim to make things work. If things get too complicated (and that is a judgement call), it's probably time to consider modifying the standard distribution of your CGI interface library and offer the modifications for general use.

Listing 3 shows the correct way to use CGIplus and fully implement server agnosticism as understood so far.

```
unshift @INC,"HT_ROOT:[SRC.PERL]" ;

use strict;
use 5.6.1 ;

my $useCGIplus = ($ENV{'CGIPLUSEOF'} ne undef)  &&
    ($ENV{'SCRIPT_RTE'} eq undef) ;
eval { require CGIplus ; } || die "Can't find CGIplus.pm" ;
if ($useCGIplus)
{
    #CGIplus::stripWWW(1);
    CGIplus::process(\&doit) ;
}
else
{
    doit()
}

sub doit
{
    require CGI if (!defined(&CGI::new)) ;
    my $theCGI = new CGI ;
    print $theCGI->header('text/plain') ;
    my @theParameterNames = $theCGI->param() ;
    my $theString = "The Parameter/Value pairs" ;
    $theString .= " (Usage: " . CGIplus::usageCount() . ")"
        if (CGIplus::isCGIplus()) ;
    $theString .= ":\n" ;
    print $theString ;
    print "-------------------------\n\n" ;
```

```
    foreach (sort @theParameterNames)
    {
        my @theValue = $theCGI->param($_) ;
        if ($#theValue)
        {
            for (my $i = 0; $i <= $#theValue; $i++)
            {
                $theString = $_ . "[$i] = " . $theValue[$i] . "\n" ;
                print $theString ;
            }
        }
        else
        {
            $theString = $_ . " = " . $theValue[0] . "\n" ;
            print $theString ;
        }
    }
    print "\n-------------------------\n" ;
    if ($ENV{'QUERY_STRING'} eq "eoj")
    {
        exit ;
    }
}
```

Listing 3
Fully Agnostic CGI

The third lesson in building server-agnostic CGIs is to ensure that the output of your CGI is constant across all environments. When http://www.csworks.com/cgiplus-bin/vtj/listing3.pl ?foo=1 is run repeatedly, and the value of the query parameter varied, the display of the CGI output remains constant, the display of the usage counter comes and goes depending on the environment (CGI or CGIplus), and those pesky extra new lines have been eliminated.

However, those pesky new lines are important if CGIs that output binary data such as graphics are written. When the standard output data stream is opened for the second and later instances of a CGI executing in the CGIplus execution environment, the standard output stream is opened in **record** mode. This means that for every I/O operation presented to the standard output stream a record delimiter is added. In this case, an extra new line (or carriage return/line feed pair, I haven't verified which), is inserted at the end of each I/O operation. Since the standard output stream isn't buffering data, each collection of data gets written in a new operation, and each operation has a record delimiter, therefore the extra lines. By collecting the data to be presented into strings, and writing each fully formatted string in a single I/O operation, formatting is preserved.

We need to briefly address providing binary data output in a server-agnostic fashion. CGIplus.pm provides interfaces that write binary data to the standard output stream. These work for all WASD Perl execution environments **except** OSU emulation, or within OSU.

As was seen above, if the standard output stream is in record mode, spurious data is introduced into the CGI output stream. For HTML data this is largely harmless, but for binary data of any kind, it's fatal. The data stream is corrupted by the spurious carriage control. The necessary fix for the OSU server and WASD OSU emulation is to remove the change to record mode for the standard output stream. The following shows the portion of WWWEXEC.COM that is changed.

```
$ perl_script:
$   tfile = "sys$scratch:perlcgi_" + f$string(f$getjpi("0","PID")) + ".tmp"
$!  write_net "<DNETRECMODE>"
$   on warning then goto perl_done
```

Once this is done, the binary output interfaces provided by WASD's CGIplus.pm can be used. Note that a "typical" Perl CGI using CGI.pm and using a standard output stream that is not in record mode will break. CGI.pm is written based on record mode being OpenVMS's default behavior for the standard output stream. In a production environment WWWEXEC.COM should be modified so that record mode is the default except for some class of CGIs that could be detected by directory, file name or file extension. Implementation of this patch at your site is left as an exercise to the reader.

For the interested reader a server-agnostic CGI that does binary output has been implemented and can be downloaded from http://www.csworks.com/download/modularian.html.

The last execution environment available to Perl using the WASD server is PerlRTE. PerlRTE is essentially a persistent Perl Interpreter capable of running any Perl CGI. The state of the CGI is discarded upon CGI exit, so PerlRTE is similar to the standard CGI execution environment without the overhead of sub-process creation and loading of the Perl interpreter. A CGIplus script cannot be activated (the module detects and prevents it) using the PerlRTE in RTE mode (a seemingly subtle but very real distinction with WASD). As seen in listing 3, CGIplus::process is only executed if CGIPLUSEOF is defined in the Perl environment and the SCRIPT_RTE environment variable is undefined. Since the SCRIPT_RTE environment variable is always defined by PerlRTE then CGIplus::process will never get executed and the CGI in listing 3 is ready to execute correctly in the PerlRTE environment.

Apache for OpenVMS

One server has gone unmentioned in this article, not because I wished to ignore it but because I can't run a copy easily (my systems are still running 7.1) without breaking out an old box and spending a couple of days building up a new system. However since these server-agnostic CGIs rely upon CGI.pm and CGI.pm is well known to function in a mod_cgi or mod_perl environment, then I believe that server-agnostic CGIs developed for WASD and OSU should also run "out of the box" on Apache as well. Mark Daniels (the author of WASD) tested listing3.pl under Apache for OpenVMS[5] for me. The script runs unmodified. This makes server-agnostic programming even more useful since by being able to run under OSU, WASD, and Apache for OpenVMS, the large majority of OpenVMS web server installations are accounted for. Platform agnostic CGIs, those written in a "portable" language (Perl, Python, PHP, et al.) and using nothing but commonly available interfaces, can move from platform to platform without change, and can thus easily be made truly agnostic, caring about neither server nor platform.

Summary

Using two of the web servers commonly available for OpenVMS systems and a simple CGI, we've deduced three guiding principles for developing server-agnostic CGIs. These principles are:

1. Make sure that the execution environment meets the expectations of your CGI interface library
2. Make sure that the innards of your CGI interface library stay sane independent of the execution environment
3. Make sure the output of the CGI stays constant independent of the execution environment

The mechanisms used to build server-agnostic CGIs are not expensive (in terms of performance) or difficult (in terms of programming); all it takes is a little care.

[5] The test configuration was OpenVMS 7.3-1, CSWS 1.3, and CPQ Perl 5.6.1 or CSWS Perl 1.1.

The benefits are that your CGIs can be used "anywhere," allowing you and your customers more flexibility in terms of development, debugging, and deployment.

I'm interested in seeing server agnostic CGIs in as many environments as possible. So far I have accommodated WASD, OSU, and Apache for OpenVMS. If there are others in active use on OpenVMS, I'd love to know what has to change to make listing3.pl work on your server. In addition, please send be the results of listing3.pl on platforms other than OpenVMS. I can be reached at munroe@csworks.com.

Thanks

I'd like to thank my reviewers and editors, Mark Daniels, author of WASD, Jennifer Cole Ripman, my wife, and Ben Ripman, my son. Their contributions to this article made it significantly better than it would otherwise have been. Any errors remaining at this point are mine and not theirs.

For more information

Modularian – A server agnostic CGI that produces binary output.

Framework – A Perl script I used to convert a client's DCL OSU CGIs to simple server agnostic CGIs.

ServerAgnosticPerl.zip – sources of the listings for this article.

FastCGI Interface Specification – The FastCGI protocol and interface specification.

CGI Interface Specification – The Common Gatewate Interface specification.

PerlEx – A Perl performance enhancement for Windows.

WASD - A web server implemented using multiple processes.

OSU – A web server implemented using DECThreads.

Afterword

Perl makes, more or less, anything possible. The listings shown in this article present an "under the hood" view of server agnostic CGIs. It is possible to package these requirements and simplify the process enormously. As an exercise I have done so. Included in ServerAgnosticPerl.zip are two additional files, listing4.pl and saperlcgi.pm which demonstrate how well hidden the details of server agnostic CGI programming can be made in Perl.