OpenVMS Technical Journal V10

Implementation of a web application maintenance and testing environment

Willem Grooters, VX Company BV, OpenVMS developer and system manager

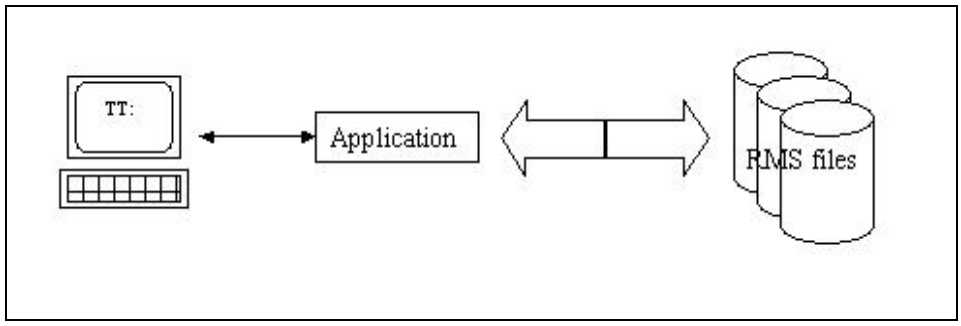
Introduction

Most locations where an application is developed and maintained have some kind of separation between the environments of code storage, actual development (the programmer) and testing. In classic environments, this is usually sufficient. Development and testing of web-based applications is more complicated because each environment requires a web server to debug and test the application. It gets even more complicated if more than one web server must be supported. This article describes how such an environment can be configured.

The application and its history

The basis for this article is an application, used by law enforcement, written in DIBOL in the 1980s, originally for VAX systems accessed on VT terminals. In the mid-1990s, the application was ported to the Alpha architecture.

The architecture of this application has been straightforward from the beginning. Data is stored in indexed files, most of them having multiple keys. The application includes just a few main images for data entry (which can be quite complex in structure and coherence) and data maintenance, and a few executables for application management:



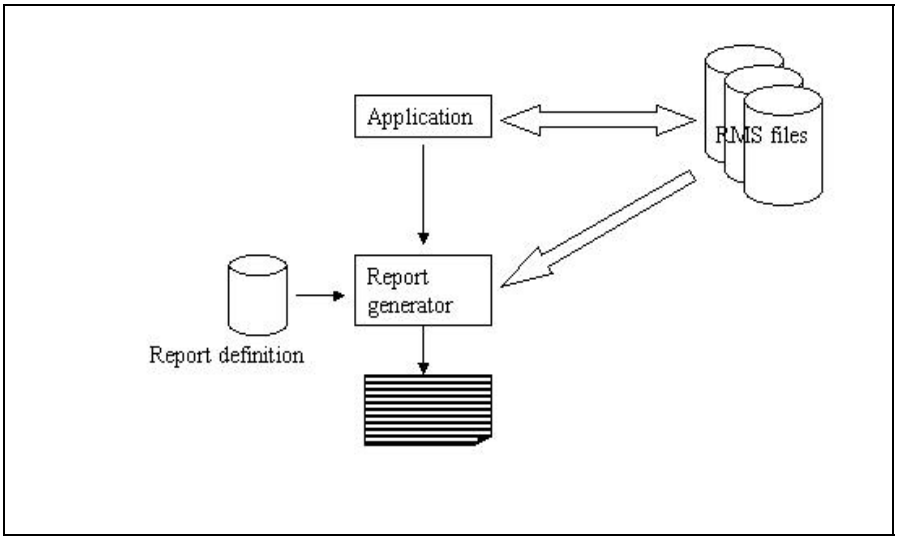
The application is maintained centrally but is used in over 20 different, largely independent locations – with their own system and application management.

Each location has its own group of users. These are end users who access the application programs by a menu system (either built in or locally maintained DCL procedures), allowing them to access application programs based on their functional profile, which is stored and maintained as part of the application. Some have access to just the basic functionality, others can access higher privileged functions, and a few -- the application managers – can access the most privileged functions and programs for maintaining data consistency, adjustment of application parameters, and a number of system management tasks.

As with any application, reports are created, some as a result of the normal functionality, some at the request of the user or application managers. Most of these reports are required for further legal activity, and some hold management information. Some are coded within the application, but localized reports were a requirement. Because of the nature of the application, some reports are subject to change as a result of government legislation – and sometimes quite drastically. Having these reports coded within the application meant a huge amount of work. For the local reports, the required customizations would have caused maintenance to become error prone or impossible.

For that reason, a report generator was introduced that is started by the application with some controlling data, to produce a report from the RMS files. The definition of the report includes layout, selections, filters, and transformations, and is coded and stored separately. These features make management of the reports easier.

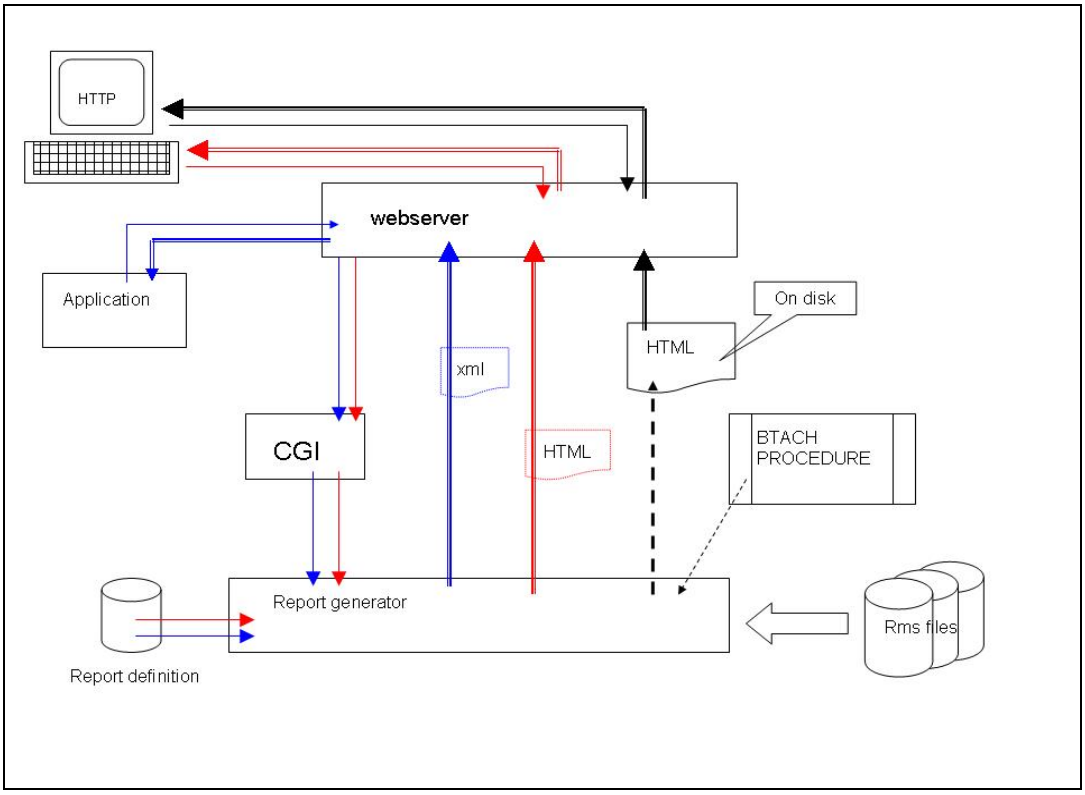
Reports can be created by the application, either as part of the workflow or in batch mode. The following figure shows the flow of data to and from the report generator.



Users of this application also needed the ability to create and modify reports interactively -- if only to test the application. Reports were developed that could be created without intervention of the application itself. Report generation could be started by a terminal session, but a batch procedure was also a possibility.

The mid-1990s saw a drive for accessing the data in a web browser. Because reports could be created in batch, it was a small step to develop report definitions that created HTML pages that were accessible by a web browser over a web server.

In the beginning, these reports were created daily in batch mode. Later, interactive access was introduced: static pages allow the user to enter search criteria, thereby launching a CGI procedure that starts the report generator. The report generator then creates pages that, in turn, can create lists and data reports and can even display reports that have been created from the application. The following figure shows this process:



Legend:
Black: access of report, generated in batch and stiored on disk
Red: access of data based on selection specified in URL,, returns data in HTML (or xhtml) format
Blue: Access of data based on selection specified in URL, returns data in XML format.
Single line: request
Double line: returned output

The original application, however, remains the only way to add and modify data and to do the main processing on the system. The web interface is used only as another means to view the data. The only exception is the possibility of uploading files that are related to existing data, but these uploaded files can be accessed only using the web interface. The data that constitutes the link, however, can be accessed and modified (by authorized personnel only) by a normal executable.

A word about the report generator

The basis of reports are report definitions, that is, plain text files containing simply a piece of code that describes what is being read from the application files, based on data that is passed by symbols or logicals, on data that is stored within the data files, or that is hardcoded in the definition. Specifically, the code describes what filters and conversions are applied to the read data and how the information is displayed on SYS$OUTPUT – to the terminal, in a file to be stored on disk or sent to a printer, or, for web requests, input to the web server that sends the output to the requesting browser.

This code is compiled into a byte-code file using indexed files that describe the application's files in terms of records, fields (their offsets, data type, and size), and keys (key number, starting position, and size). This byte code is input to the report generator to create the report.

The report generator is originally developed as part of the application, but in such a way it can be used by other applications as well. The files describing the files in the application, the report descriptions, and the resulting byte code, however, are part of the application.

Versioning

All elements of this structure -- the application and the web interface -- are closely related. The report generator is an application by itself, but it processes report definitions that are part of the application or web interface. To process these report definitions, the generator needs data about the application files to create the output. Such data includes keys, location of data files within the records, and naming of these fields, and so on. These files are specific to the application. When a file changes, as with the removal of a key or the addition of a field in the record, the files need to be updated as well.

By these generator files that describe the files, the report generator can now exactly locate the fields from a file. If any of these fields moves, the wrong data is shown. Therefore, the right files need to be addressed.

Processing the report definitions actually means compiling them into byte code to be interpreted and executed by the generator. The compiled version is specific to a given version of the application or web interface. However, in some cases, changes to the report compiler or generator require recompilation of the report definitions. For that reason, the release of a new report generator always coincides with a new release of the application and web interface. Therefore, for the web server, it is important to access the right version of both the web interface and report generator.

Application structure

It is important to realize that the development team has no control over the versions used in each of the locations. Two or three versions might be active and each needs to be maintained. Also, moving from one version to another can mean that two versions are temporarily accessible at the same time at one location, and both versions must be maintained simultaneously. Therefore, for any given activity, the user must always be aware of what version is in use.

At some locations, different departments use different data sets. That is, the files are identical in structure and name, but their content is different. Within the location these are called "administrations"; externally, the only difference is the number used in the physical name. For instance, one department can have number 001, the other 003. Thus, a file logically named HFD is named HFD001 for the first department, and a similar file for the other department is named HFD003. (This is part of the DIBOL structure used in the original environment.) The structure within the application is quite strict and is the same across all versions, all data, and all administrations.

The application root contains all versions of the application that are kept on line, and all of them have the same structure, as shown in the following example:

```
ROOT
   2002A
      ADM BIB BOB COM EXE FDL GEG LIB LST MNU REC REF RPT SRC
   2002B
      ADM BIB BOB COM EXE FDL GEG LIB LST MNU REC REF RPT SRC
   2004A
      ADM BIB BOB COM EXE FDL GEG LIB LST MNU REC REF RPT SRC
   2006A
      ADM BIB BOB COM EXE FDL GEG LIB LST MNU REC REF RPT SRC
```

The report generator has a similar structure.

This structure is used for all working environments, from programming to production, where the production environment would miss some of them – obviously the ones containing source code. Each of these directories is accessible using a single logical name that is set by a command procedure and is maintained within each version.

The web application has a similar structure but with one addition: the entry point for the web server is shared by all versions, and the web server has no data about application versions. The following example shows this structure:

```
CGI
   APPLWEB
      COM
   APPLROOT
      2002A
            BOB COM EXE REC RPT SRC
      2002B
         BOB COM EXE REC RPT SRC
      2004A
         BOB COM EXE REC RPT SRC
      2006A
         BOB COM EXE REC RPT SRC
```

Physically, the version structure of the web application is stored within the application structure and is separately rooted, as shown in the following example:

```
ROOT
   2002A
      ADM BIB BOB COM EXE FDL GEG LIB LST MNU REC REF RPT SRC WEB
                                                    BOB COM EXE REC RPT
SRC
   2002B
      ADM BIB BOB COM EXE FDL GEG LIB LST MNU REC REF RPT SRC WEB
                                                    BOB COM EXE REC RPT
SRC
   200$A
      ADM BIB BOB COM EXE FDL GEG LIB LST MNU REC REF RPT SRC WEB
                                                    BOB COM EXE REC RPT
SRC
```

The CGI directory, however, is kept outside the structure because it is part of the web server.

The production environment matches this structure based on logical names. Most users have these logicals defined in their login command files or have them defined by a menu system.

The web interface, however, is not part of the application in the production environment; rather, it is a separately installed product. The CGI procedure that is started by the web server determines a name based on application, version, administration, and server; looks for a logical with that name; and runs the corresponding command procedure. That procedure then sets all the required logical names that refer to this web interface and to the application data.

Development environment

Development and maintenance of the application is structured into a number of working environments.
The central, common environment is Development (DVLP). This is used as a container for files that are used to create a new release. There is no direct development in this environment.

Each programmer has a separate working area, following the structure described in the preceding section and referred to by the same logicals as any would be referred to, but now set up as a search list containing the programmer's own environment first, followed by the DVLP environment.

Although the DVLP environment contains the web interface as part of the application, this is not true for the programmer's environment. In the latter, the application and the web interface to it are two distinct applications, even though they share the same data files. The primary reason for this distinction is to prevent unintended interaction between the two.

For the application itself, testing in the programmer's environment does not differ from running the application in a normal way. However, the web interface cannot be tested that way. This is one of the problems to solve.

Different web servers

Various web servers are available on OpenVMS, and any of these can be used in a production environment.

The first web server that is used in all locations is the Ohio State University web server, known as the OSU web server. At the time, this was the only web server that was free and that offered sufficient capabilities and secure access. This web server relies on DECnet for its CGI scripting, and it requires the following specific commands in the CGI command procedure for the processing:

```
$ write net_link "<DNETRECMODE>"        ! Set implied carriage control.
$ mcr www_root:[bin]CGI_SYMBOLS "WWW_" "FORM_"
$!
$! Send back CGI response.  Note that newlines (<CR><LF>) must be
$! explicitly sent.
$!
```

Since DECnet was to be phased out after 2000, there is the need to switch web server – and Secure Web Server (SWS) was introduced. This is based on the well-known UNIX-based Apache server. In testing the application with this server, it was soon discovered that the CGI procedure needed to be changed (for example, all OSU-specific code needed to be removed), and that the server handled spaces and other characters very differently. Consequently, the report definitions of the web interface needed to be changed as well. Since not all locations could transition immediately, the CGI procedure for SWS was different from the one from OSU.

The reports needed to change as well. The OSU web server has no trouble with characters like space, greater than and less than, colon, and so on. However, SWS cannot handle these characters properly, so translations needed to be done. To avoid having separate code for OSU and SWS, translation requirements were identified by checking for the logical APACHE$COMMON, and if that existed, translations were executed as necessary. The end result was one code base for the report definitions but two distinct kits for the web interface to be released: one for OSU and one for SWS.

New developments were now first tested on SWS, but testing them on OSU was troublesome and soon showed that, in some cases, drastic changes were required for the new development to be usable on the OSU web server. This situation could have been prevented if testing under this server had been possible from the beginning.

Later it was discovered that SWS was not able to handle mixed-case passwords, a requirement in some locations. The WASD web server was introduced as an alternative for this reason. Research revealed that, except for a few differences, the CGI procedure as defined for SWS was usable.

At that time, a complete overhaul of the web interface was started in one programmer's environment, whereas others started new development and did regular maintenance. These activities required the configuration of a web server environment that facilitated access to different locations without interfering with access to others. It also meant that not just one web server was available, but all that applied to the organization were available.

The challenge

The issue now was how to create an environment where:

- Each programmer has a "web" to develop and test as follows:
    - Without interfering with other programmer's work
    - In a near-production structure
    - By each of the three web servers concurrently
- The version that is delivered can be accessed as follows:
    - By each web server
    - In a standard structure

The second point (versioning) was particularly important because, at that moment, no such environment existed.

The implementation

Installing different web servers is not a big problem because their root directories are different, as are their logical names. The main issue is their configuration – each should have its own set of ports to listen to.

The "reference" system – that is, the system that has been delivered – gets a port on each server. Since each programmer will use each of the three web servers, each programmer was assigned a port on each server, to be used in the URL to access the specific environment.

Added to these was the need for a port to access the web interface in the DVLP environment, and one for a test environment (the version to be delivered) to allow regression and acceptance testing on each server.

Actually, this is what "multiple webs" is all about: one web server serving a number of different, unrelated webs. All three servers support multiple webs, even though their naming differs.

It was decided that one machine – separate from DVLP and, therefore, from the programmers – would be set up for testing and research.

On the DVLP system, configuration for each web server was as follows:

- A virtual web for the reference system

- A virtual web for the DVLP system
- A virtual web for each of the developers

On the test system, the configuration contained:

- A virtual web for testing HTML interface
- A virtual web for testing application access
- At least one virtual web for research

Because of development of a common CGI procedure, and the requirement to start with renewal of the reports, we started with the development machine. Here, only SWS 1.3 was installed, and testing on the OSU web server could be done on another machine.

For maintenance reasons, the web servers were installed off the system disk under a single root directory, as shown in the following example:

```
APPL_DISK
   WEBSERVER
      OSU            ← WEB$ROOT
         OSU structure
      APACHE      ← APACHE$ROOT
            SWS structure
      HT_ROOT     ← HT_ROOT:
```

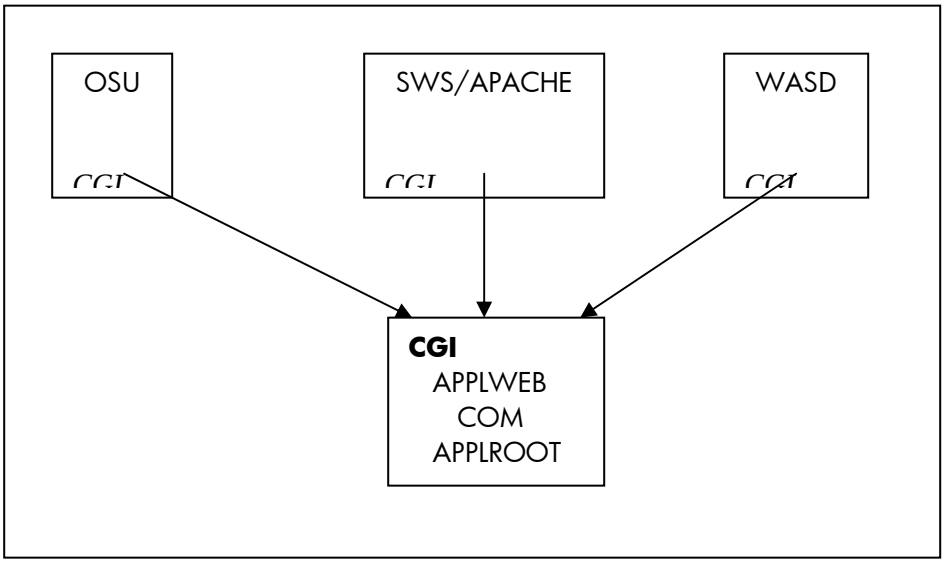
The next point was to map the structures for the web server in such a way that it would be identical for each of them: Each web server needed to see the identical structure. That meant only a single CGI procedure could handle all requests. Another requirement was that the URL used could not contain any data that was specific for one server, implying that any differences would need to be handled by the CGI procedure. Furthermore, changing the URL was out of the question.

The mapping of the structure meant correct definition of logical names. This was already present in both the OSU and SWS web server environments in the production environment, and so needed to be set up in the DVLP and the programmers' environments. Because a single command procedure was to handle all three servers, this issue was moved to the development of that procedure.

First the structure that is immediately accessed by the web server (that is, the CGI directory and the content below it) was to be set up. Because the access of the web interface requires authentication on first access, the standard [CGI-BIN] directory cannot be used, in stead CGI is a different directory outside the server structure: it is located in the environment it belongs to. The same applies to the web-interface files – however, these are physically located in the application environment as described earlier. The logical names however give it a look as a different application, as if it was a production environment.

This works fine for one version, but if different versions were to be accessed, a different CGI directory tree would be required for each version of the application. This does not conform to the production environment – one CGI procedure would have to be used for all versions. (The URL contains all the needed data.) Changing the configuration is not an acceptable solution because the process is error prone, and because it might disrupt existing connections by requiring the server to be restarted.

The solution was found by creating an alias for the directory into each of the web server roots (SET FILE /ENTER). That way, only one physical CGI directory tree is used by all three the web servers, as shown in the following figure:



Any changes anywhere in this directory tree can be directly accessed by any web server, so the whole environment can be tested in parallel. Access to this central location has become part of the procedures used by the deveolpment team.

On this system, just one environment was foreseen initially, so this structure was examined in that environment first. Each web server served on its own port: OSU on port 80 (because that was found to be the hardest to configure), SWS on port 82, and WASD on port 84.

We installed a very simple web application on the CGI directory – just a few HTML files and one procedure – as a proof of concept to show this method would work. After that, we started converting the existing SWS configuration on the development machine. The version that was last delivered was already installed on port 80, and we kept it there. Its configuration, however, was moved to the first "Virtual Host" entry in the configuration, as shown in the following example:

```
### Section 3: Virtual Hosts
## defined ports
## ------------------
# NameVirtualhost aaa.bbb.ccc.ddd
include conf/virthost.conf

## DEFAULT (port 80)
<VirtualHost _default_ aaa.bbb.ccc.ddd>
    ServerAdmin xxxxxx
    Documentroot "/apache$common/cgi
    ErrorLog logs/error_log
    CustomLog logs/access_log common
<Directory "/apache$common/cgi">
    Options Indexes FollowSymLinks Multiviews
    AllowOverride All
    Order allow,deny
    Allow from all
    AuthType Basic
      Authname "Live system"
    AuthUserOpenVMS on
    AuthAuthoritative On
    require valid-user
</Directory>
</VirtualHost>
```

Ports 81 to 90 were also mentioned as ports to listen to, and for each port a configuration file was created to be included in the server's configuration file. These files would hold the information of the programmer's location, so each programmer had a separate web.

Each programmer was to have a separate CGI directory tree that would resemble the live version I structure and files – including the CGI procedure used. Starting with the current SWS-related procedure, only one change was needed: where the live CGI procedure creates a logical to access the local definition of working environment, the setup for the programmer's environment could be served by a fixed one. Otherwise, a number of logicals would have to be defined, one for each programmer and this is not needed because the structure is well defined in this environment. This single procedure actually mimics the user's login when setting up the programmer's development area, causing the application logicals to be defined like the programmer would have them defined. That is, the logicals refer first to the programmer's directory, and second, to the DVLP area. Any procedure or image running would now be run as if the programmer started it, as shown in the following example:

```
## Port 86
##
## (WILLEM)
##
<VirtualHost aaa.bbb.ccc.ddd: 86>
    DocumentRoot /user/willem/cgi
    ErrorLog logs/willem-error_log
    CustomLog logs/willem-access_log common

<Directory "/user/willem/cgi">
    Options Indexes FollowSymLinks Multiviews
    AllowOverride All
    Order allow,deny
    Allow from all
    AuthType Basic
    Authname "WILLEM"
    AuthUserOpenVMS on
    AuthAuthoritative On
    require valid-user
</Directory>

# Scripting is version independent
ScriptAlias /cgi/ "/user/willem/cgi/appl/com/"

# Application web interface files version 2004
Alias /appl/04/WebIF/ "/user/willem/WebIF04/"
Alias /appl/04/ "/user/willem/WebIF04/"

# Application web interface files version 2004
Alias /appl/06/WebIF/ "/user/willem/WebIF06/"
Alias /appl/06/ "/user/willem/WebIF06/"

</VirtualHost>
```

In this case, the programmer is working on two different versions of the application.

The next step was development of a single CGI procedure to access the web interface, regardless of the web server used. In the current configuration, two web servers were used: one for OSU and one for SWS. These two needed to be joined to form one procedure that was easily adapted to use with WASD (and any other web server). One big advantage with the procedures for OSU and SWS web servers was the significant similarity between them: The code examining and interpreting the URL to set up the environment, to do some consistency and sanity checks, and actual running the report generator were only minor issues. These parts were extracted and placed in separate command files that were then called by the main procedure. For development and testing, a few other command files were created, one of them showing all symbols that are set up by the web server.

Major differences that showed up are based on the difference in servers, and first could be partly solved by determining the existence of the logicals that refered the root directories of the server. That would be fine if either one or the other server was running. However, in this construction, all three servers were active, so using these logicals was not an option. The first possible solution was to define a logical outside the server specifying the server used, but that was abandoned because it was not feasible in this test configuration. In a live system, this solution would require the intervention of a system manager or operator in case the server was switched --- a task that is easily overlooked.

In the end, the solution was to use process information. Since the CGI procedure is run in a subprocess of the server, the name is derived from it. For SWS, its name would contain the string “APACHE”, and for WASD, the name would contain the string “HTTPD”. If neither is true, the server is OSU. That way, some primary setup can be done, as shown in the following example:

```
$       HTTP_SERVER==f$edit(f$GETJPI("","PRCNAM"),"UPCASE")
$       if f$locate("APACHE", HTTP_SERVER) .ne. f$length(HTTP_SERVER)
$       then
$               HTTP_SERVER == "APACHE"
$       else
$               if f$locate("HTTPD", HTTP_SERVER) .ne. f$length(HTTP_SERVER)
$               then
$                       HTTP_SERVER == "WASD"
$               else
$                       if f$locate("HTTP", HTTP_SERVER) .ne. f$legth(HTTP_SERV
$                       then
$                               wrnet "<DNETRECMODE>"  ! Set implied carriage c
$                               mcr www_root:[bin]CGI_SYMBOLS "WWW_" "FORM_"
$                               HTTP_SERVER == "OSU"
$                               URI == P2
$                       else
$                               ! --- foutje-------
$                               err = "Ophalen soort server APACHE/OSU mislukt."
$                               err1 = "Check logical BPS$SERVER_SOFTWARE."
$                               @gate:gate_alert "HTML_TAG"
$                               goto closeMe
$                       endif
$               endif
$       endif
```

The type of web server is also important in the retrieval of server data. For instance, the OSU web server uses a different method than SWS or WASD. For OSU, the URL up to the question mark (“?”) is passed as a second parameter to the CGI procedure, the rest as a symbol. So the full URL needs to be contructed. On the other hand, SWS and WASD get the full URL by symbols set by the server, but the names differ.

For WASD, we found another difference: data embedded in a <FORM> was found to be named differently than the way SWS names the symbols. To be able to retrieve these arguments, an extra translation was found to be necessary, as shown in the following example:

```
$ if HTTP_SERVER .eqs. "WASD"
$ then
$!
$       WWW_ANR                 == "''anr'"
$       if f$type (WWW_FORM_BIJLAGE) .nes. "" then WWW_FLD_BIJLAGE == "''fld_bijlage'"
$       if f$type (WWW_FORM_SOORT) .nes. "" then WWW_FLD_SOORT == "''fld_soort'"
$       if f$type (WWW_FORM_BIJLAGE) .nes. "" then WWW_FLD_KEY == "''fld_key'"
$ endif
```

Otherswise, the program that needs these values is unable to locate them (this image used the same library that is included with WASD, and it worked with both SWS and WASD), as shown in the following example:

```
$ if HTTP_SERVER .eqs. "APACHE"
$ then
$       fullHostName = SERVER_NAME
$       URL_STRING = REQUEST_URI
$       RemoteUser = REMOTE_USER
$!
$ else
$       if HTTP_SERVER .eqs. "OSU"
$       then
$               fullHostName = WWW_SERVER_NAME
$       url_string="''URI'?''WWW_QUERY_STRING'"
$               RemoteUser= WWW_REMOTE_USER
$       else
$               if HTTP_SERVER .eqs. "WASD"
$               then
$                       fullHostName = WWW_SERVER_NAME
$                       URL_STRING = WWW_REQUEST_URI
$                       RemoteUser = WWW_REMOTE_USER
$               else
$                   err = "Unknown HTTP_Server."
$                   err1 = "CHECH SERVER SOFTWARE
$                   @gate:gate_alert
$                   goto exit
$               endif
$       endif
$ endif
```

The same symbol is used to do some preparation of the output.

The preparation for HTML output was found to be different in original procedures. The OSU-based procedure did not contain anything particular for this, but the SWS procedure has the following defined:

```
$ write sys$output f$fao("!/<HTML>")
$!
```

This definition seemed to be a requirement, and removing it caused a “Server Error”, even when coded inside a report. The definition was also found to be a requirement when using the internally present debug mode output. This could again be solved by executing this code only when required, and bypassing it when not applicable.

For XML output, the page needed to be prepared differently for OSU and SWS as well. Both need to have the header defined properly but in a different way, regardless of the server, as shown in the following example:

```
$write sys$output f$fao("Content-Type: text/html")
$ write sys$output f$fao
("!/<!!DOCTYPE html PUBLIC '-//W3C//DTDXHTML 1.0 Strict//EN' 'DTD/xhtml1-strict.dtd' >")
$ write sys$output f$fao
("!/<html xmlns='http://www.w3.org/1999/xhtml' xml:lang='nl' lang='nl'>")
```

The OSU web server requires that the processing is "closed down" before continuing. Again, this is done based on the symbol HTTP_SERVER, as shown in the following example:

```
$ if HTTP_SERVER .eqs. "OSU"
$ then
$    wrnet "content-type: text/html"      ! CGI header
$    wrnet "status: 200 exe ready"
$    wrnet "extra-header: EXEC info, system: ", system, ", prog: ", prog
$    wrnet ""
$ endif
```

With some other server-specific processing, the server-independent CGI procedure was completed and could be used for processing plain HTML, generated reports, execution of executables that produce and handle forms, as well as for displaying any type of data.

However, WASD was found to have one more drawback: If the disk accessed was an ODS-5 disk, it explicitly changed the process parse style to "extended". This caused problems with the report generator, which didn't recognize options passed in lowercase. Investigations revealed that the routine that retrieved the options from the command line used the LIB$GET_FOREIGN library function to retrieve the options and parameters, and checked the occurrence of its parameters – after converting the passed arguments to uppercase. With traditional parsing, LIB$GET_FOREIGN converts the read lowercase data to uppercase, so expected command-line options are indeed found. With an extended parseing style, however, LIB$GET_FOREIGN does not perform this transformation, so the same lowercase option is not recognized. This problem was solved simply by setting the parsing style to "traditional" for all servers.

Once this procedure was running, the translation of special characters in the URL data, as well as the HTML and XML output, needed to be reviewed. This was the only change to be made to the report definitions. Originally, the existence of the logical APACHE$COMMON was sufficient to determine whether translations were required or not, but again, in this configuration it was not applicable. Because the symbol HTTP_SERVER has been set up by the CGI procedure based on the calling server, we can now use that symbol to perform, or bypass, the conversions.

```
$defvar #http_server, symbol
…
$if #http_server = "APACHE" $or #http_server = "WASD"  $then
$translate #url "%20"  "  "
$endif
```

The programmers were now able to change files in their own environments without intefering with other programmers' work, and the redesign of the web interface could be performed separately from other activities.

Once this was set up, work began on the second system. The work was hastened by the breakdown of the one OSU server, which required a new server to be configured. This system was to be used as a demonstration, test, and research system. Only one environment would be active at a time, which made the configuration somewhat less complex.

Again, the web servers are installed off the system disk, in their own directory, as on the development system. However, the root directories are one level deeper, since it should be possible to have multiple versions of a server at hand – though only one version of a particular server can be active at a time, as shown in the following example:

```
APPL_DISK
   WEBSERVER
      OSU
         3.10          ← WEB$ROOT
           OSU structure
      APACHE
          1.3
           SWS structure
          2.1           ← APACHE$ROOT
           SWS structure
          2.1-1
           SWS structure
      HT_ROOT
          9. 1
            WASD Structure
           9.2           ← HT_ROOT:
            WASD Structure
```

In this example, OSU version 3.10, SWS version 2.1 and WASD version 9.2 are active.

In this way, the active version can be easily switched, by stopping the server, redefining the logical referring the root directory, and restarting the server.

Installation off the system disk is required on this system because of its nature: that is, it is destined to be used to test the application on new versions of OpenVMS, as well as to test the web interface under new versions of the web servers. One of the system's disks is kept free for testing alone, so installing a new OpenVMS version would not require a reinstallation of server software, and switching between versions would be as easy as rebooting.

Like the development system, the CGI directory tree is located outside the web server structure, but a link is made in each of them, so here as well, the CGI directory seems to reside directly below the web server. That allows the internal structure of the web interface to be exactly the same as on the development system – no matter from what environment it is copied. (If different environments were required, the construction within each of the web servers would be similar as the configuration on the development system. This however has not been installed at this moment.)

The CGI procedure that was created on the development machine was copied to this machine and tested with all three servers. Changes were applied on the development system and were copied back afterward. Once the procedure was finalized, both the application and the web interface were copied to this machine.

Conclusion

Setting up an environment where multiple developers work concurrently on web applications and are able to debug and test them in a natural way -- using a web server – is not a time-consuming task in itself. Given the abilities of the web servers themselves (virtual hosts on different ports) and of OpenVMS (logicals to define multiple locations, creating aliases for directories), it is fairly simple. Assigning ports and a principal configuration of the default web on each server take the majority of time. Setting up a system with multiple, concurrent web servers is no problem either, as long as each one runs on its own port.

Developing a single, all-purpose CGI procedure for an application that works regardless of the web server used requires more work and requires knowledge of each web server's method of naming symbols, the way it handles logicals and special characters, and, of course, of the web application's requirements. For this study, having two very similar procedures that were already working was a huge advantage; starting from scratch would have caused far more work. Finally, the book by Alan Winston titled *OpenVMS with Apache, OSU and WASD: The Nonstop Webserver,* although an older volume, was still very helpful in our testing scenarios.