CHAPTER 22

Troubleshooting mod_perl

When something goes wrong, we expect the software to report the problem. But if we don't understand the meaning of the error message, we won't be able to resolve it. Therefore in this chapter we will talk about errors specific to mod_perl, as reported by a mod_perl-enabled Apache server.

Many reports are produced by Perl itself. If you find them unclear, you may want to use the `use diagnostics` pragma in your development code. With the `diagnostics` pragma, Perl provides an in-depth explanation of each reported warning and error. Note that you should remove this pragma in your production code, since it adds a runtime overhead.

Errors that may occur during the build and installation stages are covered in the respective troubleshooting sections of Chapter 3. This chapter deals with errors that may occur during the configuration and startup, code parsing and compilation, runtime, and shutdown and restart phases.

Configuration and Startup

This section covers errors you might encounter when you start the server.

libexec/libperl.so: open failed: No such file or directory

If you get this error when you start the server, it probably means that your version of Perl was itself compiled with a shared library called *libperl.so*. mod_perl detects this and links the Apache executable to the same Perl shared library. This error simply means that the shared library cannot be found by searching the paths that Apache knows about.

Make sure you have Perl installed on the machine, and that you have *libperl.so* in `<perlroot>/<version>/<architecture>/CORE` (for example, */usr/local/lib/perl5/5.6.1/sun4-solaris/CORE*).

If the file is there but you still get the error, you should include the directory in which the file is located in the environment variable LD_LIBRARY_PATH (or the equivalent variable for your operating system). Under normal circumstances, Apache should have had the library path configured properly at compile time; if Apache was misconfigured, adding the path to LD_LIBRARY_PATH manually will help Apache find the shared library.

install_driver(Oracle) failed: Can't load '.../DBD/Oracle/Oracle.so' for module DBD::Oracle

Here's an example of the full error report that you might see:

```
install_driver(Oracle) failed: Can't load
'/usr/lib/perl5/site_perl/5.6.1/i386-linux/auto/DBD/Oracle/Oracle.so'
for module DBD::Oracle:
libclntsh.so.8.0: cannot open shared object file:
No such file or directory at
/usr/lib/perl5/5.6.1/i386-linux/DynaLoader.pm line 169.
at (eval 27) line 3
Perhaps a required shared
library or dll isn't installed where expected at
/usr/local/apache/perl/tmp.pl line 11
```

On BSD-style filesystems, LD_LIBRARY_PATH is not searched for *setuid* programs. If Apache is a *setuid* executable, you might receive this error. Therefore, the first solution is to explicitly load the library from the system-wide *ldconfig* configuration file:

```
panic# echo $ORACLE_HOME/lib >> /etc/ld.so.conf
panic# ldconfig
```

Another solution to this problem is to modify the *Makefile* file (which is created when you run *perl Makefile.PL*) as follows:

1. Search for the line LD_RUN_PATH=
2. Replace it with LD_RUN_PATH=my_oracle_home/lib

where *my_oracle_home* is, of course, the home path to your Oracle installation. In particular, the file *libclntsh.so.8.0* should exist in the *lib* subdirectory.

Then just type *make install*, and all should go well.

Note that setting LD_RUN_PATH has the effect of hardcoding the path to *my_oracle_home/lib* in the file *Oracle.so*, which is generated by DBD::Oracle. This is an efficiency mechanism, so that at runtime it doesn't have to search through LD_LIBRARY_PATH or the default directories used by *ld*.

For more information, see the *ld* manpage and the essay on LD_LIBRARY_PATH at *http://www.visi.com/~barr/ldpath.html*.

Invalid command 'PerlHandler'...

Here's an example of the full error report that you might see:

```
Syntax error on line 393 of /home/httpd/httpd_perl/conf/httpd.conf:
Invalid command 'PerlHandler', perhaps mis-spelled or
defined by a module not included in the server
configuration [FAILED]
```

You might get this error when you have a mod_perl-enabled Apache server compiled with DSO, but the mod_perl module isn't loaded. (This generally happens when it's an installed RPM or other binary package.) In this case you have to tell Apache to load mod_perl by adding the following line to your *httpd.conf* file:

```
AddModule mod_perl.c
```

You might also get this error when you try to run a non-mod_perl Apache server using the *httpd.conf* file from a mod_perl server.

RegistryLoader: Translation of uri [...] to filename failed

Here's an example of the full error report that you might see:

```
RegistryLoader: Translation of uri
  [/home/httpd/perl/test.pl] to filename failed
  [tried: /home/httpd/docs/home/httpd/perl/test.pl]
```

In this example, this means you are trying to preload a script called */perl/test.pl*, located at */home/httpd/perl/test.pl* in the filesystem. This error shows up when `Apache::RegistryLoader` fails to translate the URI into the corresponding filesystem path. Most failures happen when a user passes a file path (such as */home/httpd/perl/test.pl*) instead of a relative URI (such as */perl/test.pl*).

You should either provide both the URI and the filename:

```
Apache::RegistryLoader->new->handler($uri, $filename);
```

or supply a callback subroutine that will perform the URI-to-filename conversion. The callback accepts the URI as an argument and returns a filename. For example, if your mod_perl scripts reside in */home/httpd/perl-scripts/* but the base URI is */perl/*, you might do the following:

```
my $rl = Apache::RegistryLoader->new(
             trans => \&uri2filename);
$rl->handler("/perl/test.pl");

sub uri2filename{
    my $uri = shift;
    $uri =~ s:^/perl/:/perl-scripts/:;
    return Apache->server_root_relative($uri);
}
```

Here, we initialize the `Apache::RegistryLoader` object with the `uri2filename()` function that will perform the URI-to-filename translation. In this function, we just adjust

the URI and return the filename based on the location of the server root. So if the server root is */home/httpd/*, the callback will return */home/httpd/perl-scripts/test.pl*—exactly what we have requested.

For more information please refer to the `Apache::RegistryLoader` manpage.

Code Parsing and Compilation

The following warnings and errors might be reported when the Perl code is compiled. This may be during the server startup phase or, if the code hasn't yet been compiled, at request time.

Value of $x will not stay shared at - line 5

This warning usually happens when scripts are run under `Apache::Registry` and similar handlers, and some function uses a lexically scoped variable that is defined outside of that function.

This warning is important and should be considered an error in most cases. The explanation of the problem and possible solutions are discussed in Chapter 6.

Value of $x may be unavailable at - line 5

Similar to the previous section, the warning may happen under `Apache::Registry` and similar handlers, and should be considered an error. The cause is discussed in the *perldiag* manpage and possible solutions in Chapter 6.

Can't locate loadable object for module ...

Here's an example of the full error report that you might see:

```
Can't locate loadable object for module Apache::Util in @INC...
```

In this particular example, it means that there is no object built for `Apache::Util`. You should build mod_perl with one of these arguments: `PERL_UTIL_API=1`, `EVERYTHING=1`, or `DYNAMIC=1`.

For similar errors, see Chapter 3. Locate the missing module and see what build-time argument enables it.

Can't locate object method "get_handlers" ...

If you see this error:

```
Can't locate object method "get_handlers" via package "Apache"
```

you need to rebuild your mod_perl with stacked handlers; that is, with `PERL_STACKED_HANDLERS=1` or with `EVERYTHING=1`.

Missing right bracket at line ...

This error usually means you really do have a syntax error. However, you might also see it because a script running under `Apache::Registry` is using either the `__DATA__` or `__END__` tokens. In Chapter 6, we explain why this problem arises when a script is run under `Apache::Registry`.

Can't load '.../auto/DBI/DBI.so' for module DBI

If you have the `DBI` module installed, this error is usually caused by binary incompatibilities. Check that all your modules were compiled with the same Perl version that mod_perl was built with. For example, Perl 5.005 and 5.004 are not binary compatible by default.

Other known causes of this problem are:

- OS distributions that ship with a broken binary Perl installation.
- The `perl` program and `libperl.a` library are somehow built with different binary compatibility flags.

The solution to these problems is to rebuild Perl and any extension modules from a fresh source tree. Read Perl's *INSTALL* document for more details.

On the Solaris OS, if you see the "Can't load DBI" or a similar error for the `IO` module (or whatever dynamic module mod_perl tries to pull in first), you need to reconfigure, rebuild, and reinstall Perl and any dynamic modules. When *Configure* asks for "additional LD flags," add the following flags:

```
-Xlinker --export-dynamic
```

or:

```
-Xlinker -E
```

This problem is known to be caused only by installing GNU *ld* under Solaris.

Runtime

Once you have your server up and running and most of the code working correctly, you may still encounter errors generated by your code at runtime. Some possible errors are discussed in this section.

foo ... at /dev/null line 0

Under mod_perl, you may receive a warning or an error in the *error_log* file that specifies */dev/null* as the source file and line 0 as the line number where the printing of the message was triggered. This is quite normal if the code is executed from within

a handler, because there is no actual file associated with the handler. Therefore, `$0` is set to */dev/null*, and that's what you see.

Segfaults When Using XML::Parser

If some processes have segmentation faults when using `XML::Parser`, you should use the following flags during Apache configuration:

```
--disable-rule=EXPAT
```

This should be necessary only with mod_perl Version 1.22 and lower. Starting with mod_perl Version 1.23, the `EXPAT` option is disabled by default.

exit signal Segmentation fault (11)

If you build mod_perl and mod_php in the same binary, you might get a segmentation fault followed by this error:

```
exit signal Segmentation fault (11)
```

The solution is to not rely on PHP's built-in MySQL support, and instead build mod_php with your local MySQL support files by adding *--with-mysql=/path/to/mysql* during *./configure*.

CGI Code Is Returned as Plain Text Instead of Being Executed

If the CGI program is not actually executed but is just returned as plain text, it means the server doesn't recognize it as a CGI script. Check your configuration files and make sure that the `ExecCGI` option is turned on. For example, your configuration section for `Apache::Registry` scripts should look like this:

```
<Location /perl>
    SetHandler perl-script
    PerlHandler Apache::Registry
    Options +ExecCGI
</Location>
```

rwrite returned -1

This error message is returned when the client breaks the connection while your script is trying to write to the client. With Apache 1.3.x, you should see the `rwrite` messages only if `LogLevel` is set to `debug`. (Prior to mod_perl 1.19_01, there was a bug that reported this debug message regardless of the value of the `LogLevel` directive.)

Generally `LogLevel` is either `debug` or `info`. `debug` logs everything, and `info` is the next level, which doesn't include debug messages. You shouldn't use `debug` mode on a production server. At the moment there is no way to prevent users from aborting connections.

Global symbol "$foo" requires explicit package name

This error message is printed when a nondeclared variable is used in the code running under the `strict` pragma. For example, consider the short script below, which contains a `use strict;` pragma and then shamelessly violates it:

```
#!/usr/bin/perl -w
use strict;
print "Content-type: text/html\n\n";
print "Hello $username";
```

Since Perl will insist that all variables are defined before being used, the program will not run and will print the error:

```
Global symbol "$username" requires
explicit package name at /home/httpd/perl/tmp.pl line 4.
```

Moreover, in certain situations (e.g., when `SIG{__DIE__}` is set to `Carp::confess()`) the entire script is printed to the *error_log* file as code that the server has tried to evaluate, so if this script is run repeatedly, the *error_log* file will grow very fast and you may run out of disk space.

This problem can easily be avoided by always declaring variables before using them. Here is the fixed version of our example:

```
#!/usr/bin/perl -w
use strict;
my $username = '';
print "Content-type: text/html\n\n";
print "Hello $username";
```

Use of uninitialized value at (eval 80) line 12

If you see this message, your code includes an undefined variable that you have used as if it was already defined and initialized. For example:

```
my $param = $q->param('test');
print $param;
```

You can fix this fairly painlessly by just specifying a default value:

```
my $param = $q->param('test') || '';
print $param;
```

In the second case, `$param` will always be defined, either with `$q->param('test')`'s return value or the default value the empty string (`''` in our example).

Undefined subroutine &Apache::ROOT::perl::test_2epl:: some_function called at

This error usually happens when two scripts or handlers (`Apache::Registry` in this case) call a function defined in a library without a package definition, or when the

two use two libraries with different content but an identical name (as passed to `require()`).

Chapter 6 provides in-depth coverage of this conundrum and numerous solutions.

Callback called exit

"Callback called exit" is just a generic message when Perl encounters an unrecoverable error during `perl_call_sv()`. mod_perl uses `perl_call_sv()` to invoke all handler subroutines. Such problems seem to occur far less often with Perl Version 5.005_03 than 5.004. It shouldn't appear with Perl Version 5.6.1 and higher.

Sometimes you discover that your server is not responding and its *error_log* file has filled up the remaining space on the filesystem. When you finally get to see the contents of the *error_log* file, it includes millions of lines like this:

```
Callback called exit at -e line 33, <HTML> chunk 1.
```

This is because Perl can get very confused inside an infinite loop in your code. It doesn't necessarily mean that your code called `exit()`. It's possible that Perl's `malloc()` went haywire and called `croak()`, but no memory was left to properly report the error, so Perl gets stuck in a loop writing that same message to `STDERR`.

Perl Version 5.005 and higher is recommended for its improved *malloc.c*, and also for other features that improve the performance of mod_perl and are turned on by default.

See also the next section.

Out of memory!

If something goes really wrong with your code, Perl may die with an "Out of memory!" and/or "Callback called exit" message. Common causes of this are infinite loops, deep recursion, or calling an undefined subroutine.

If `-DPERL_EMERGENCY_SBRK` is defined, running out of memory need not be a fatal error: a memory pool can be allocated by using the special variable `$^M`. See the *perlvar* manpage for more details.

If you compile with that option and add `use Apache::Debug level => 4;` to your Perl code, it will allocate the `$^M` emergency pool and the `$SIG{__DIE__}` handler will call `Carp::confess()`, giving you a stack trace that should reveal where the problem is. See the `Apache::Resource` module for the prevention of spinning *httpd*s.

Note that Perl 5.005 and later have `PERL_EMERGENCY_SBRK` turned on by default.

Another trick is to have a startup script initialize `Carp::confess()`, like this:

```
use Carp ();
eval { Carp::confess("init") };
```

This way, when the real problem happens, `Carp::confess` doesn't eat memory in the emergency pool (`$^M`).

syntax error at /dev/null line 1, near "line arguments:"

If you see an error of this kind:

```
syntax error at /dev/null line 1, near "line arguments:"
Execution of /dev/null aborted due to compilation errors.
parse: Undefined error: 0
```

there is a chance that your */dev/null* device is broken. You can test it with:

```
panic% echo > /dev/null
```

It should silently complete the command. If it doesn't, */dev/null* is broken. Refer to your OS's manpages to learn how to restore this device. On most Unix flavors, this is how it's done:

```
panic# rm /dev/null
panic# mknod /dev/null c 1 3
panic# chmod a+rw /dev/null
```

You need to create a special file using *mknod*, for which you need to know the type and both the major and minor modes. In our case, `c` stands for character device, `1` is the major mode, and `3` is the minor mode. The file should be readable and writable by everybody, hence the permission mode settings (`a+rw`).

Shutdown and Restart

When you shut down or restart the server, you may encounter the problems presented in the following sections.

Evil Things Might Happen When Using PerlFreshRestart

Unfortunately, not all Perl modules are robust enough to survive reload. For them this is an unusual situation. `PerlFreshRestart` does not much more than:

```
while (my($k,$v) = each %INC) {
    delete $INC{$k};
    require $k;
}
```

Besides that, it flushes the `Apache::Registry` cache and empties any dynamic stacked handlers (e.g., `PerlChildInitHandler`).

Lots of segfaults and other problems have been reported by users who turned on `PerlFreshRestart`. Most of them go away when it is turned off. It doesn't mean that you shouldn't use `PerlFreshRestart`, if it works for you. Just beware of the dragons.

Note that if you have a mod_perl-enabled Apache built as a DSO and you restart it, the whole Perl interpreter is completely torn down (via `perl_destruct()`) and restarted. The value of `PerlFreshRestart` is irrelevent at this point.

[warn] child process 30388 did not exit, sending another SIGHUP

With Apache Version 1.3.0 and higher, mod_perl will call the `perl_destruct()` Perl API function during the child exit phase. This will cause proper execution of any `END` blocks found during server startup and will also invoke the `DESTROY` method on global objects that still exist.

It is possible that this operation will take a long time to finish, causing problems during a restart. If you use the *apachectl* script to restart the server, it sends the `SIGHUP` signal after waiting for a short while. The `SIGHUP` can cause problems, since it might disrupt something you need to happen during server shutdown (for example, saving data).

If you are certain that your code does not contain any `END` blocks or `DESTROY` methods to be run during child server shutdown, you can avoid the delays by setting the `PERL_DESTRUCT_LEVEL` environment variable to `-1`. Be careful, however; even if your code doesn't include any `END` blocks or `DESTROY` methods, any modules you `use()` might.

Processes Get Stuck on Graceful Restart

If after doing a graceful restart (e.g, by sending *kill -USR1*) you see via mod_status or `Apache::VMonitor` that a process is stuck in state *G* (Gracefully finishing), it means that the process is hanging in `perl_destruct()` while trying to clean up. If you don't need the cleanup, see the previous section on how to disable it.