1 Apache::Reload - Reload Perl Modules when Changed on Disk

1.1 Synopsis

```
# Monitor and reload all modules in %INC:
# httpd.conf:
PerlModule Apache::Reload
PerlInitHandler Apache::Reload

# when working with protocols and connection filters
# PerlPreConnectionHandler Apache::Reload

# Reload groups of modules:
# httpd.conf:
PerlModule Apache::Reload
PerlInitHandler Apache::Reload
PerlSetVar ReloadAll Off
PerlSetVar ReloadModules "ModPerl::* Apache::*"
#PerlSetVar ReloadDebug On
#PerlSetVar ReloadConstantRedefineWarnings Off

# Reload a single module from within itself:
package My::Apache::Module;
use Apache::Reload;
sub handler { ... }
1;
```

1.2 Description

`Apache::Reload` reloads modules that change on the disk.

When Perl pulls a file via `require`, it stores the filename in the global hash `%INC`. The next time Perl tries to `require` the same file, it sees the file in `%INC` and does not reload from disk. This module's handler can be configured to iterate over the modules in `%INC` and reload those that have changed on disk or only specific modules that have registered themselves with `Apache::Reload`. It can also do the check for modified modules, when a special touch-file has been modified.

Note that `Apache::Reload` operates on the current context of `@INC`. Which means, when called as a `Perl*Handler` it will not see `@INC` paths added or removed by `Apache::Registry` scripts, as the value of `@INC` is saved on server startup and restored to that value after each request. In other words, if you want `Apache::Reload` to work with modules that live in custom `@INC` paths, you should modify `@INC` when the server is started. Besides, `'use lib'` in the startup script, you can also set the `PERL5LIB` variable in the httpd's environment to include any non-standard 'lib' directories that you choose. For example, to accomplish that you can include a line:

```
PERL5LIB=/home/httpd/perl/extra; export PERL5LIB
```

in the script that starts Apache. Alternatively, you can set this environment variable in *httpd.conf*:

```
PerlSetEnv PERL5LIB /home/httpd/perl/extra
```

1.2.1 Monitor All Modules in %INC

To monitor and reload all modules in `%INC` at the beginning of request's processing, simply add the following configuration to your *httpd.conf*:

```
PerlModule Apache::Reload
PerlInitHandler Apache::Reload
```

When working with connection filters and protocol modules `Apache::Reload` should be invoked in the pre_connection stage:

```
PerlPreConnectionHandler Apache::Reload
```

See also the discussion on `PerlPreConnectionHandler`.

1.2.2 Register Modules Implicitly

To only reload modules that have registered with `Apache::Reload`, add the following to the *httpd.conf*:

```
PerlModule Apache::Reload
PerlInitHandler Apache::Reload
PerlSetVar ReloadAll Off
# ReloadAll defaults to On
```

Then any modules with the line:

```
use Apache::Reload;
```

Will be reloaded when they change.

1.2.3 Register Modules Explicitly

You can also register modules explicitly in your *httpd.conf* file that you want to be reloaded on change:

```
PerlModule Apache::Reload
PerlInitHandler Apache::Reload
PerlSetVar ReloadAll Off
PerlSetVar ReloadModules "My::Foo My::Bar Foo::Bar::Test"
```

Note that these are split on whitespace, but the module list **must** be in quotes, otherwise Apache tries to parse the parameter list.

The `*` wild character can be used to register groups of files under the same namespace. For example the setting:

```
PerlSetVar ReloadModules "ModPerl::* Apache::*"
```

will monitor all modules under the namespaces `ModPerl::` and `Apache::`.

1.2.4 Monitor Only Certain Sub Directories

To reload modules only in certain directories (and their subdirectories) add the following to the *httpd.conf*:

```
PerlModule Apache::Reload
PerlInitHandler Apache::Reload
PerlSetVar ReloadDirectories "/tmp/project1 /tmp/project2"
```

You can further narrow the list of modules to be reloaded from the chosen directories with `ReloadModules` as in:

```
PerlModule Apache::Reload
PerlInitHandler Apache::Reload
PerlSetVar ReloadDirectories "/tmp/project1 /tmp/project2"
PerlSetVar ReloadAll Off
PerlSetVar ReloadModules "MyApache::*"
```

In this configuration example only modules from the namespace `MyApache::` found in the directories */tmp/project1/* and */tmp/project2/* (and their subdirectories) will be reloaded.

1.2.5 Special "Touch" File

You can also declare a file, which when gets `touch(1)`ed, causes the reloads to be performed. For example if you set:

```
PerlSetVar ReloadTouchFile /tmp/reload_modules
```

and don't `touch(1)` the file */tmp/reload_modules*, the reloads won't happen until you go to the command line and type:

```
% touch /tmp/reload_modules
```

When you do that, the modules that have been changed, will be magically reloaded on the next request. This option works with any mode described before.

1.3 Performance Issues

This modules is perfectly suited for a development environment. Though it's possible that you would like to use it in a production environment, since with `Apache::Reload` you don't have to restart the server in order to reload changed modules during software updates. Though this convenience comes at a price:

- If the "touch" file feature is used, `Apache::Reload` has to stat(2) the touch file on each request, which adds a slight but most likely insignificant overhead to response times. Otherwise `Apache::Reload` will stat(2) each registered module or even worse--all modules in `%INC`, which will significantly slow everything down.

- Once the child process reloads the modules, the memory used by these modules is not shared with the parent process anymore. Therefore the memory consumption may grow significantly.

Therefore doing a full server stop and restart is probably a better solution.

1.4 Debug

If you aren't sure whether the modules that are supposed to be reloaded, are actually getting reloaded, turn the debug mode on:

```
PerlSetVar ReloadDebug On
```

1.5 Silencing 'Constant subroutine ... redefined at' Warnings

If a module defines constants, e.g.:

```
use constant PI => 3.14;
```

and gets re-loaded, Perl issues a mandatory warnings which can't be silenced by conventional means (since Perl 5.8.0). This is because constants are inlined at compile time, so if there are other modules that are using constants from this module, but weren't reloaded they will see different values. Hence the warning is mandatory. However chances are that most of the time you won't modify the constant subroutine and you don't want *error_log* to be cluttered with (hopefully) irrelevant warnings. In such cases, if you haven't modified the constant subroutine, or you know what you are doing, you can tell `Apache::Reload` to shut those for you (it overrides `$SIG{__WARN__}` to accomplish that):

```
PerlSetVar ReloadConstantRedefineWarnings Off
```

For the reasons explained above this option is turned on by default.

since: mod_perl 1.99_10

1.6 Caveats

1.6.1 Problems With Reloading Modules Which Do Not Declare Their Package Name

If you modify modules, which don't declare their `package`, and rely on `Apache::Reload` to reload them, you may encounter problems: i.e., it'll appear as if the module wasn't reloaded when in fact it was. This happens because when `Apache::Reload` `require()`s such a module all the global symbols end up in the `Apache::Reload` namespace! So the module does get reloaded and you see the compile time errors if there are any, but the symbols don't get imported to the right namespace. Therefore the old version of the code is running.

1.6.2 Problems with Scripts Running with Registry Handlers that Cache the Code

The following problem is relevant only to registry handlers that cache the compiled script. For example it concerns `ModPerl::Registry` but not `ModPerl::PerlRun`.

1.6.2.1 The Problem

Let's say that there is a module `My::Utils`:

```
#file:My/Utils.pm
#----------------
package My::Utils;
BEGIN { warn __PACKAGE__ , " was reloaded\n" }
use base qw(Exporter);
@EXPORT = qw(colour);
sub colour { "white" }
1;
```

And a registry script *test.pl*:

```
#file:test.pl
#------------
use My::Utils;
print "Content-type: text/plain\n\n";
print "the color is " . colour();
```

Assuming that the server is running in a single mode, we request the script for the first time and we get the response:

```
the color is white
```

Now we change *My/Utils.pm*:

```
-  sub colour { "white" }
+  sub colour { "red" }
```

And issue the request again. `Apache::Reload` does its job and we can see that `My::Utils` was reloaded (look in the *error_log* file). However the script still returns:

```
the color is white
```

1.6.2.2 The Explanation

Even though *My/Utils.pm* was reloaded, `ModPerl::Registry`'s cached code won't run '`use My::Utils;`' again (since it happens only once, i.e. during the compile time). Therefore the script doesn't know that the subroutine reference has been changed.

This is easy to verify. Let's change the script to be:

```
#file:test.pl
#------------
use My::Utils;
print "Content-type: text/plain\n\n";
my $sub_int = \&colour;
my $sub_ext = \&My::Utils::colour;
print "int $sub_int\n";
print "ext $sub_ext\n";
```

Issue a request, you will see something similar to:

```
int CODE(0x8510af8)
ext CODE(0x8510af8)
```

As you can see both point to the same CODE reference (meaning that it's the same symbol). After modifying *My/Utils.pm* again:

```
-  sub colour { "red" }
+  sub colour { "blue" }
```

and calling the script on the secondnd time, we get:

```
int CODE(0x8510af8)
ext CODE(0x851112c)
```

You can see that the internal CODE reference is not the same as the external one.

1.6.2.3 The Solution

There are two solutions to this problem:

Solution 1: replace `use()` with an explicit `require()` + `import()`.

```
- use My::Utils;
+ require My::Utils; My::Utils->import();
```

now the changed functions will be reimported on every request.

Solution 2: remember to touch the script itself every time you change the module that it requires.

1.7 Threaded MPM and Multiple Perl Interpreters

If you use `Apache::Reload` with a threaded MPM and multiple Perl interpreters, the modules will be reloaded by each interpreter as they are used, not every interpreters at once. Similar to mod_perl 1.0 where each child has its own Perl interpreter, the modules are reloaded as each child is hit with a request.

If a module is loaded at startup, the syntax tree of each subroutine is shared between interpreters (big win), but each subroutine has its own padlist (where lexical my variables are stored). Once `Apache::Reload` reloads a module, this sharing goes away and each Perl interpreter will have its own copy of the syntax tree for the reloaded subroutines.

1.8 Pseudo-hashes

The short summary of this is: Don't use pseudo-hashes. They are deprecated since Perl 5.8 and are removed in 5.9.

Use an array with constant indexes. Its faster in the general case, its more guaranteed, and generally, it works.

The long summary is that some work has been done to get this module working with modules that use pseudo-hashes, but it's still broken in the case of a single module that contains multiple packages that all use pseudo-hashes.

So don't do that.

1.9 Copyright

1.10 Authors

Matt Sergeant, matt@sergeant.org

Stas Bekman (porting to mod_perl 2.0)

A few concepts borrowed from `Stonehenge::Reload` by Randal Schwartz and `Apache::StatINC` (mod_perl 1.x) by Doug MacEachern and Ask Bjoern Hansen.

1.11 See Also

`Stonehenge::Reload`

Table of Contents:

1 Apache::Reload - Reload Perl Modules when Changed on Disk 1
1.1 Synopsis 2
1.2 Description 2
1.2.1 Monitor All Modules in %INC 3
1.2.2 Register Modules Implicitly 3
1.2.3 Register Modules Explicitly 3
1.2.4 Monitor Only Certain Sub Directories 4
1.2.5 Special "Touch" File 4
1.3 Performance Issues 4
1.4 Debug 5
1.5 Silencing 'Constant subroutine ... redefined at' Warnings 5
1.6 Caveats 5
1.6.1 Problems With Reloading Modules Which Do Not Declare Their Package Name . . 5
1.6.2 Problems with Scripts Running with Registry Handlers that Cache the Code . . . 6
1.6.2.1 The Problem 6
1.6.2.2 The Explanation 6
1.6.2.3 The Solution 7
1.7 Threaded MPM and Multiple Perl Interpreters 7
1.8 Pseudo-hashes 8
1.9 Copyright 8
1.10 Authors 8
1.11 See Also 8