1   Porting Apache:: Perl Modules from mod_perl 1.0 to 2.0

1.1 Description

This document describes the various options for porting a mod_perl 1.0 Apache module so that it runs on a Apache 2.0 / mod_perl 2.0 server. It's also helpful to those who start developing mod_perl 2.0 handlers.

Developers who need to port modules using XS code, should also read about porting Apache:: XS modules.

There is also: Porting CPAN modules to mod_perl 2.0 Status.

1.2 Introduction

In the vast majority of cases, a perl Apache module that runs under mod_perl 1.0 will **not** run under mod_perl 2.0 without at least some degree of modification.

Even a very simple module that does not in itself need any changes will at least need the mod_perl 2.0 Apache modules loaded, because in mod_perl 2.0 basic functionality, such as access to the request object and returning an HTTP status, is not found where, or implemented how it used to be in mod_perl 1.0.

Most real-life modules will in fact need to deal with the following changes:

- methods that have moved to a different (new) package

- methods that must be called differently (due to changed prototypes)

- methods that have ceased to exist (functionality provided in some other way)

Do not be alarmed! One way to deal with all of these issues is to load the `Apache::compat` compatibility layer bundled with mod_perl 2.0. This magic spell will make almost any 1.0 module run under 2.0 without further changes. It is by no means the solution for every case, however, so please read carefully the following discussion of this and other options.

There are three basic options for porting. Let's take a quick look at each one and then discuss each in more detail.

1. **Run the module on 2.0 under `Apache::compat` with no further changes**

   As we have said mod_perl 2.0 ships with a module, `Apache::compat`, that provides a complete drop-in compatibility layer for 1.0 modules. `Apache::compat` does the following:

   - Loads all the mod_perl 2.0 Apache:: modules

   - Adjusts method calls where the prototype has changed

   - Provides Perl implementation for methods that no longer exist in 2.0

The drawback to using `Apache::compat` is the performance hit, which can be significant.

Authors of CPAN and other publicly distributed modules should not use `Apache::compat` since this forces its use in environments where the administrator may have chosen to optimize memory use by making all code run natively under 2.0.

2. **Modify the module to run only under 2.0**

   If you are not interested in providing backwards compatibility with mod_perl 1.0, or if you plan to leave your 1.0 module in place and develop a new version compatible with 2.0, you will need to make changes to your code. How significant or widespread the changes are depends largely of course on your existing code.

   Several sections of this document provide detailed information on how to rewrite your code for mod_perl 2.0 Several tools are provided to help you, and it should be a relatively painless task and one that you only have to do once.

3. **Modify the module so that it runs under both 1.0 and 2.0**

   You need to do this if you want to keep the same version number for your module, or if you distribute your module on CPAN and want to maintain and release just one codebase.

   This is a relatively simple enhancement of option (2) above. The module tests to see which version of mod_perl is in use and then executes the appropriate method call.

The following sections provide more detailed information and instructions for each of these three porting strategies.

1.3 Using `Apache::porting`

META: to be written. this is a new package which makes chunks of this doc simpler. for now see the `Apache::porting` manpage.

1.4 Using the `Apache::compat` Layer

The `Apache::compat` module tries to hide the changes in API prototypes between version 1.0 and 2.0 of mod_perl, and implements "virtual methods" for the methods and functions that actually no longer exist.

`Apache::compat` is extremely easy to use. Either add at the very beginning of startup.pl:

```
use Apache2;
use Apache::compat;
```

or add to httpd.conf:

```
PerlModule Apache2
PerlModule Apache::compat
```

That's all there is to it. Now you can run your 1.0 module unchanged.

Remember, however, that using `Apache::compat` will make your module run slower. It can create a larger memory footprint than you need and it implements functionality in pure Perl that is provided in much faster XS in mod_perl 1.0 as well as in 2.0. This module was really designed to assist in the transition from 1.0 to 2.0. Generally you will be better off if you port your code to use the mod_perl 2.0 API.

It's also especially important to repeat that `CPAN module developers are requested not to use this module in their code`, since this takes the control over performance away from users.

1.5 Porting a Perl Module to Run under mod_perl 2.0

Note: API changes are listed in the mod_perl 1.0 backward compatibility document.

The following sections will guide you through the steps of porting your modules to mod_perl 2.0.

1.5.1 Using *ModPerl::MethodLookup* to Discover Which mod_perl 2.0 Modules Need to Be Loaded

It would certainly be nice to have our mod_perl 1.0 code run on the mod_perl 2.0 server unmodified. So first of all, try your luck and test the code.

It's almost certain that your code won't work when you try, however, because mod_perl 2.0 splits functionality across many more modules than version 1.0 did, and you have to load these modules before the methods that live in them can be used. So the first step is to figure out which these modules are and `use()` them.

The `ModPerl::MethodLookup` module provided with mod_perl 2.0 allows you to find out which module contains the functionality you are looking for. Simply provide it with the name of the mod_perl 1.0 method that has moved to a new module, and it will tell you what the module is.

For example, let's say we have a mod_perl 1.0 code snippet:

```
$r->content_type('text/plain');
$r->print("Hello cruel world!");
```

If we run this, mod_perl 2.0 will complain that the method `content_type()` can't be found. So we use `ModPerl::MethodLookup` to figure out which module provides this method. We can just run this from the command line:

```
% perl -MApache2 -MModPerl::MethodLookup -e print_method content_type
```

This prints:

```
to use method 'content_type' add:
        use Apache::RequestRec ();
```

We do what it says and add this `use()` statement to our code, restart our server (unless we're using `Apache::Reload`), and mod_perl will no longer complain about this particular method.

Since you may need to use this technique quite often you may want to `define an alias`. Once defined the last command line lookup can be accomplished with:

```
% lookup content_type
```

`ModPerl::MethodLookup` also provides helper functions for finding `which methods are defined in a given module`, or `which methods can be invoked on a given object`.

1.5.1.1 Handling Methods Existing In More Than One Package

Some methods exists in several classes. For example this is the case with the `print()` method. We know the drill:

```
% lookup print
```

This prints:

```
There is more than one class with method 'print'
try one of:
        use Apache::RequestIO ();
        use Apache::Filter ();
```

So there is more than one package that has this method. Since we know that we call the `print()` method with the `$r` object, it must be the `Apache::RequestIO` module that we are after. Indeed, loading this module solves the problem.

1.5.1.2 Using `ModPerl::MethodLookup` Programmatically

The issue of picking the right module, when more than one matches, can be resolved when using `ModPerl::MethodLookup` programmatically -- `lookup_method` accepts an object as an optional second argument, which is used if there is more than one module that contains the method in question. `ModPerl::MethodLookup` knows that `Apache::RequestIO` and and `Apache::Filter` expect an object of type `Apache::RequestRec` and type `Apache::Filter` respectively. So in a program running under mod_perl we can call:

```
ModPerl::MethodLookup::lookup_method('print', $r);
```

Now only one module will be matched.

This functionality can be used in AUTOLOAD, for example, although most users will not have a need for this robust of solution.

1.5.1.3 Pre-loading All mod_perl 2.0 Modules

Now if you use a wide range of methods and functions from the mod_perl 1.0 API, the process of finding all the modules that need to be loaded can be quite frustrating. In this case you may find the function preload_all_modules() to be the right tool for you. This function preloads **all** mod_perl 2.0 modules, implementing their API in XS.

While useful for testing and development, it is not recommended to use this function in production systems. Before going into production you should remove the call to this function and load only the modules that are used, in order to save memory.

CPAN module developers should **not** be tempted to call this function from their modules, because it prevents the user of their module from optimizing her system's memory usage.

1.5.2 Handling Missing and Modified mod_perl 1.0 Methods and Functions

The mod_perl 2.0 API is modeled even more closely upon the Apache API than was mod_perl version 1.0. Just as the Apache 2.0 API is substantially different from that of Apache 1.0, therefore, the mod_perl 2.0 API is quite different from that of mod_perl 1.0. Unfortunately, this means that certain method calls and functions that were present in mod_perl version 1.0 are missing or modified in mod_perl 2.0.

If mod_perl 2.0 tells you that some method is missing and it can't be found using ModPerl::Method-Lookup, it's most likely because the method doesn't exist in the mod_perl 2.0 API. It's also possible that the method does still exist, but nevertheless it doesn't work, since its usage has changed (e.g. its prototype has changed, or it requires different arguments, etc.).

In either of these cases, refer to the backwards compatibility document for an exhaustive list of API calls that have been modified or removed.

1.5.2.1 Methods that No Longer Exist

Some methods that existed in mod_perl 1.0 simply do not exist anywhere in version 2.0 and you must therefore call a different method o methods to get the functionality you want.

For example, suppose we have a mod_perl 1.0 code snippet:

```
$r->log_reason("Couldn't open the session file: $@");
```

If we try to run this under mod_perl 2.0 it will complain about the call to log_reason(). But when we use ModPerl::MethodLookup to see which module to load in order to call that method, nothing is found:

```
% perl -MApache2 -MModPerl::MethodLookup -le \
  'print((ModPerl::MethodLookup::lookup_method(shift))[0])' \
  log_reason
```

This prints:

```
don't know anything about method 'log_reason'
```

Looks like we are calling a non-existent method! Our next step is to refer to the backwards compatibility document, wherein we find that as we suspected, the method `log_reason()` no longer exists, and that instead we should use the other standard logging functions provided by the `Apache::Log` module.

1.5.2.2 Methods Whose Usage Has Been Modified

Some methods still exist, but their usage has been modified, and your code must call them in the new fashion or it will generate an error. Most often the method call requires new or different arguments.

For example, say our mod_perl 1.0 code said:

```
$parsed_uri = Apache::URI->parse($r, $r->uri);
```

This code causes mod_perl 2.0 to complain first about not being able to load the method `parse()` via the package Apache::URI. We use the tools described above to discover that the package containing our method has moved and change our code to load and use `APR::URI`:

```
$parsed_uri = APR::URI->parse($r, $r->uri);
```

But we still get an error. It's a little cryptic, but it gets the point across:

```
p is not of type APR::Pool at /path/to/OurModule.pm line 9.
```

What this is telling us is that the method `parse` requires an APR::Pool object as its first argument. (Some methods whose usage has changed emit more helpful error messages prefixed with "Usage: ...") So we change our code to:

```
$parsed_uri = APR::URI->parse($r->pool, $r->uri);
```

and all is well in the world again.

1.5.3 *Requiring a specific mod_perl version.*

To require a module to run only under 2.0, simply add:

```
use Apache2;
use mod_perl 2.0;
```

META: In fact, before 2.0 is released you really have to say:

```
  use Apache2;
  use mod_perl 1.99;
```

And you can even require a specific version (for example when a certain API has been added only starting from that version). For example to require version 1.99_08, you can say:

```
  use mod_perl 1.9908;
```

1.5.4 Should the Module Name Be Changed?

If it is not possible to make your code run under both mod_perl versions (see below), you will have to maintain two separate versions of your own code. While you can change the name of the module for the new version, it's best to try to preserve the name and use some workarounds.

Let's say that you have a module `Apache::Friendly` whose release version compliant with mod_perl 1.0 is 1.57. You keep this version on CPAN and release a new version, 2.01, which is compliant with mod_perl 2.0 and preserves the name of the module. It's possible that a user may need to have both versions of the module on the same machine. Since the two have the same name they obviously cannot live under the same tree.

One attempt to solve this problem is to use *Makefile.PL*'s `MP_INST_APACHE2` option. If the module is configured as:

```
  % perl Makefile.PL MP_INST_APACHE2=1
```

it'll be installed relative to the *Apache2/* directory.

META: but of course this won't work in non-core mod_perl, since a generic `Makefile.PL` has no idea what to do about `MP_INST_APACHE2=1`. Need to provide copy-n-paste recipe for this. Or even add to the core a supporting module that will handle this functionality.

The second step is to change the documentation of your 2.0 compliant module to instruct users to `use Apache2 ();` in their code (or in *startup.pl* or via `PerlModule Apache2` in *httpd.conf*) before the module is required. This will cause `@INC` to be modified to include the *Apache2/* directory first.

The introduction of the *Apache2/* directory is similar to how Perl installs its modules in a version specific directory. For example:

```
  lib/5.7.1
  lib/5.7.2
```

1.5.5 Using `Apache::compat` As a Tutorial

Even if you have followed the recommendation and eschewed use of the `Apache::compat` module, you may find it useful to learn how the API has been changed and how to modify your own code. Simply look at the `Apache::compat` source code and see how the functionality should be implemented in mod_perl 2.0.

For example, mod_perl 2.0 doesn't provide the `Apache->gensym` method. As we can see if we look at the `Apache/compat.pm` source, the functionality is now available via the core Perl module `Symbol` and its `gensym()` function. (Since mod_perl 2.0 works only with Perl versions 5.6 and higher, and `Symbol.pm` is included in the core Perl distribution since version 5.6.0, there was no reason to keep providing `Apache->gensym`.)

So if the original code looked like:

```
my $fh = Apache->gensym;
open $fh, $file or die "Can't open $file: $!";
```

in order to port it mod_perl 2.0 we can write:

```
my $fh = Symbol::gensym;
open $fh, $file or die "Can't open $file: $!";
```

Or we can even skip loading `Symbol.pm`, since under Perl version 5.6 and higher we can just do:

```
open my $fh, $file or die "Can't open $file: $!";
```

1.5.6 How `Apache::MP3` was Ported to mod_perl 2.0

`Apache::MP3` is an elaborate application that uses a lot of mod_perl API. After porting it, I have realized that if you go through the notes or even better try to do it by yourself, referring to the notes only when in trouble, you will most likely be able to port any other mod_perl 1.0 module to run under mod_perl 2.0. So here the log of what I have done while doing the porting.

Please notice that this tutorial should be considered as-is and I'm not claiming that I have got everything polished, so if you still find problems, that's absolutely OK. What's important is to try to learn from the process, so you can attack other modules on your own.

I've started to work with `Apache::MP3` version 3.03 which you can retrieve from Lincoln's CPAN directory: http://search.cpan.org/CPAN/authors/id/L/LD/LDS/Apache-MP3-3.03.tar.gz Even though by the time you'll read this there will be newer versions available it's important that you use the same version as a starting point, since if you don't, the notes below won't make much sense.

1.5.6.1 Preparations

First of all, I scratched most of mine *httpd.conf* and *startup.pl* leaving the bare minimum to get mod_perl started. This is needed to ensure that once I've completed the porting, the module will work correct on other users systems. For example if my *httpd.conf* and *startup.pl* were loading some other modules, which in turn may load modules that a to-be-ported module may rely on, the ported module may work for me, but once released, it may not work for others. It's the best to create a new *httpd.conf* when doing the porting putting only the required bits of configuration into it.

1.5.6.1.1 *httpd.conf*

Next, I configure the `Apache::Reload` module, so we don't have to constantly restart the server after we modify `Apache::MP3`. In order to do that add to *httpd.conf*:

```
PerlModule Apache::Reload
PerlInitHandler Apache::Reload
PerlSetVar ReloadAll Off
PerlSetVar ReloadModules "ModPerl::* Apache::*"
PerlSetVar ReloadConstantRedefineWarnings Off
```

You can refer to `the Apache::Reload manpage` for more information if you aren't familiar with this module. The part:

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

tells `Apache::Reload` to monitor only modules in the `ModPerl::` and `Apache::` namespaces. So `Apache::MP3` will be monitored. If your module is named `Foo::Bar`, make sure to include the right pattern for the `ReloadModules` directive. Alternatively simply have:

```
PerlSetVar ReloadAll On
```

which will monitor all modules in `%INC`, but will be a bit slower, as it'll have to `stat(3)` many more modules on each request.

Finally, `Apache::MP3` uses constant subroutines. Because of that you will get lots of warnings every time the module is modified, which I wanted to avoid. I can safely shut those warnings off, since I'm not going to change those constants. Therefore I've used the setting

```
PerlSetVar ReloadConstantRedefineWarnings Off
```

If you do change those constants, refer to the section on `ReloadConstantRedefineWarnings` .

Next I configured `Apache::MP3`. In my case I've followed the `Apache::MP3` documentation, created a directory *mp3/* under the server document root and added the corresponding directives to *httpd.conf*.

Now my *httpd.conf* looked like this:

```
#file:httpd.conf
#---------------
Listen 127.0.0.1:8002
#... standard Apache configuration bits omitted ...

LoadModule perl_module modules/mod_perl.so

PerlSwitches -wT

PerlRequire "/home/httpd/2.0/perl/startup.pl"

PerlModule Apache::Reload
PerlInitHandler Apache::Reload
PerlSetVar ReloadAll Off
```

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

AddType audio/mpeg     mp3 MP3
AddType audio/playlist m3u M3U
AddType audio/x-scpls  pls PLS
AddType application/x-ogg ogg OGG
<Location /mp3>
  SetHandler perl-script
  PerlResponseHandler Apache::MP3
  PerlSetVar PlaylistImage playlist.gif
  PerlSetVar StreamBase http://localhost:8002
  PerlSetVar BaseDir /mp3
</Location>
```

1.5.6.1.2 startup.pl

Since chances are that no mod_perl 1.0 module will work out of box without at least preloading some modules, I've enabled the `Apache::compat` module. Now my *startup.pl* looked like this:

```
#file:startup.pl
#---------------
use Apache2 ();
use lib qw(/home/httpd/2.0/perl);
use Apache::compat;
```

1.5.6.1.3 Apache/MP3.pm

Before I even started porting `Apache::MP3`, I've added the warnings pragma to *Apache/MP3.pm* (which wasn't there because mod_perl 1.0 had to work with Perl versions prior to 5.6.0, which is when the `warnings` pragma was added):

```
#file:apache_mp3_prep.diff
--- Apache/MP3.pm.orig 2003-06-03 18:44:21.000000000 +1000
+++ Apache/MP3.pm      2003-06-03 18:44:47.000000000 +1000
@@ -4,2 +4,5 @@
 use strict;
+use warnings;
+no warnings 'redefine';  # XXX: remove when done with porting
+
```

From now on, I'm going to use unified diffs which you can apply using `patch(1)`. Though you may have to refer to its manpage on your platform since the usage flags may vary. On linux I'd apply the above patch as:

```
% cd ~/perl/blead-ithread/lib/site_perl/5.9.0/
% patch -p0 < apache_mp3_prep.diff
```

(note: I've produced the above patch and one more below with `diff -u1`, to avoid the RCS Id tag geting into this document. Normally I produce diffs with `diff -u` which uses the default context of 3.)

assuming that *Apache/MP3.pm* is located in the directory *~/perl/blead-ithread/lib/site_perl/5.9.0/*.

I've enabled the `warnings` pragma even though I did have warnings turned globally in *httpd.conf* with:

```
PerlSwitches -wT
```

it's possible that some badly written module has done:

```
$^W = 0;
```

without localizing the change, affecting other code. Also notice that the *taint* mode was enabled from *httpd.conf*, something that you shouldn't forget to do.

I have also told the `warnings` pragma not to complain about redefined subs via:

```
no warnings 'redefine';  # XXX: remove when done with porting
```

I will remove that code, once porting is completed.

At this point I was ready to start the porting process and I have started the server.

```
% hup2
```

I'm using the following aliases to save typing:

```
alias err2      "tail -f ~/httpd/prefork/logs/error_log"
alias acc2      "tail -f ~/httpd/prefork/logs/access_log"
alias stop2     "~/httpd/prefork/bin/apachectl stop"
alias start2    "~/httpd/prefork/bin/apachectl start"
alias restart2  "~/httpd/prefork/bin/apachectl restart"
alias graceful2 "~/httpd/prefork/bin/apachectl graceful"
alias hup2 "stop2; sleep 3; start2; err2"
```

(I also have a similar set of aliases for mod_perl 1.0)

1.5.6.2 Porting with `Apache::compat`

I have configured my server to listen on port 8002, so I issue a request http://localhost:8002/mp3/ in one console:

```
% lynx --dump http://localhost:8002/mp3/
```

keeping the *error_log* open in the other:

```
% err2
```

which expands to:

```
% tail -f ~/httpd/prefork/logs/error_log
```

When the request is issued, the *error_log* file tells me:

```
[Thu Jun 05 15:29:45 2003] [error] [client 127.0.0.1]
Usage: Apache::RequestRec::new(classname, c, base_pool=NULL)
at .../Apache/MP3.pm line 60.
```

Looking at the code:

```
58: sub handler ($$) {
59:   my $class = shift;
60:   my $obj = $class->new(@_) or die "Can't create object: $!";
```

The problem is that handler wasn't invoked as method, but had `$r` passed to it (we can tell because `new()` was invoked as `Apache::RequestRec::new()`, whereas it should have been `Apache::MP3::new()`. Why *Apache::MP3* wasn't passed as the first argument? I go to the mod_perl 1.0 backward compatibility document and find that method handlers are now marked using the *method* subroutine attribute. So I modify the code:

```
--- Apache/MP3.pm.0     2003-06-05 15:29:19.000000000 +1000
+++ Apache/MP3.pm       2003-06-05 15:38:41.000000000 +1000
@@ -55,7 +55,7 @@
 my $NO  = '^(no|false)$';  # regular expression
 my $YES = '^(yes|true)$';  # regular expression

-sub handler ($$) {
+sub handler : method {
    my $class = shift;
    my $obj = $class->new(@_) or die "Can't create object: $!";
    return $obj->run();
```

and issue the request again (no server restart needed).

This time we get a bunch of looping redirect responses, due to a bug in mod_dir which kicks in to handle the existing dir and messing up with `$r->path_info` keeping it empty at all times. I thought I could work around this by not having the same directory and location setting, e.g. by moving the location to be */songs/* while keeping the physical directory with mp3 files as *$DocumentRoot/mp3/*, but `Apache::MP3` won't let you do that. So a solution suggested by Justin Erenkrantz is to simply shortcut that piece of code with:

```
--- Apache/MP3.pm.1     2003-06-06 14:50:59.000000000 +1000
+++ Apache/MP3.pm       2003-06-06 14:51:11.000000000 +1000
@@ -253,7 +253,7 @@
    my $self = shift;
    my $dir = shift;

-  unless ($self->r->path_info){
+  unless ($self->r->path_info eq ''){
      #Issue an external redirect if the dir isn't tailed with a '/'
      my $uri = $self->r->uri;
      my $query = $self->r->args;
```

which is equivalent to removing this code, until the bug is fixed (it was still there as of Apache 2.0.46). But the module still works without this code, because if you issue a request to */mp3* (w/o trailing slash) mod_dir, will do the redirect for you, replacing the code that we just removed. In any case this got me past this problem.

Since I have turned on the warnings pragma now I was getting loads of *uninitialized value* warnings from `$r->dir_config()` whose return value were used without checking whether they are defined or not. But you'd get them with mod_perl 1.0 as well, so they are just an example of not-so clean code, not really a relevant obstacle in my pursuit to port this module to mod_perl 2.0. Unfortunately they were cluttering the log file so I had to fix them. I've defined several convenience functions:

```
sub get_config {
    my $val = shift->r->dir_config(shift);
    return defined $val ? $val : '';
}

sub config_yes { shift->get_config(shift) !~ /$YES/oi; }
sub config_no  { shift->get_config(shift) !~ /$NO/oi; }
```

and replaced them as you can see in this patch: *code/apache_mp3_2.diff*:

```
--- Apache/MP3.pm.2     2003-06-06 15:17:22.000000000 +1000
+++ Apache/MP3.pm       2003-06-06 15:16:21.000000000 +1000
@@ -55,6 +55,14 @@
 my $NO  = '^(no|false)$';  # regular expression
 my $YES = '^(yes|true)$';  # regular expression

+sub get_config {
+    my $val = shift->r->dir_config(shift);
+    return defined $val ? $val : '';
+}
+
+sub config_yes { shift->get_config(shift) !~ /$YES/oi; }
+sub config_no  { shift->get_config(shift) !~ /$NO/oi; }
+
 sub handler : method {
   my $class = shift;
   my $obj = $class->new(@_) or die "Can't create object: $!";
@@ -70,7 +78,7 @@
   my @lang_tags;
   push @lang_tags,split /,\s+/,$r->header_in('Accept-language')
     if $r->header_in('Accept-language');
-  push @lang_tags,$r->dir_config('DefaultLanguage') || 'en-US';
+  push @lang_tags,$new->get_config('DefaultLanguage') || 'en-US';

   $new->{'lh'} ||=
     Apache::MP3::L10N->get_handle(@lang_tags)
@@ -343,7 +351,7 @@
    my $file = $subr->filename;
    my $type = $subr->content_type;
    my $data = $self->fetch_info($file,$type);
-   my $format = $self->r->dir_config('DescriptionFormat');
+   my $format = $self->get_config('DescriptionFormat');
    if ($format) {
       $r->print('#EXTINF:' , $data->{seconds} , ',');
```

```
      (my $description = $format) =~ s{%([atfglncrdmsqS%])}
@@ -1204,7 +1212,7 @@
 # get fields to display in list of MP3 files
 sub fields {
   my $self = shift;
-  my @f = split /\W+/,$self->r->dir_config('Fields');
+  my @f = split /\W+/,$self->get_config('Fields');
   return map { lc $_  } @f if @f;          # lower case
   return qw(title artist duration bitrate); # default
 }
@@ -1340,7 +1348,7 @@
 sub get_dir {
   my $self = shift;
   my ($config,$default) = @_;
-  my $dir = $self->r->dir_config($config) || $default;
+  my $dir = $self->get_config($config) || $default;
   return $dir if $dir =~ m!^/!;       # looks like a path
   return $dir if $dir =~ m!^\w+://!;  # looks like a URL
   return $self->default_dir . '/' . $dir;
@@ -1348,22 +1356,22 @@

 # return true if downloads are allowed from this directory
 sub download_ok {
-  shift->r->dir_config('AllowDownload') !~ /$NO/oi;
+  shift->config_no('AllowDownload');
 }

 # return true if streaming is allowed from this directory
 sub stream_ok {
-  shift->r->dir_config('AllowStream') !~ /$NO/oi;
+  shift->config_no('AllowStream');
 }

 # return true if playing locally is allowed
 sub playlocal_ok {
-  shift->r->dir_config('AllowPlayLocally') =~ /$YES/oi;
+  shift->config_yes('AllowPlayLocally');
 }

 # return true if we should check that the client can accomodate streaming
 sub check_stream_client {
-  shift->r->dir_config('CheckStreamClient') =~ /$YES/oi;
+  shift->config_yes('CheckStreamClient');
 }

 # return true if client can stream
@@ -1378,48 +1386,48 @@

 # whether to read info for each MP3 file (might take a long time)
 sub read_mp3_info {
-  shift->r->dir_config('ReadMP3Info') !~ /$NO/oi;
+  shift->config_no('ReadMP3Info');
 }

 # whether to time out streams
 sub stream_timeout {
-  shift->r->dir_config('StreamTimeout') || 0;
```

```
+  shift->get_config('StreamTimeout') || 0;
 }

 # how long an album list is considered so long we should put buttons
 # at the top as well as the bottom
-sub file_list_is_long { shift->r->dir_config('LongList') || 10 }
+sub file_list_is_long { shift->get_config('LongList') || 10 }

 sub home_label {
    my $self = shift;
-   my $home = $self->r->dir_config('HomeLabel') ||
+   my $home = $self->get_config('HomeLabel') ||
      $self->x('Home');
    return lc($home) eq 'hostname' ? $self->r->hostname : $home;
 }

 sub path_style {  # style for the path to parent directories
-   lc(shift->r->dir_config('PathStyle')) || 'staircase';
+   lc(shift->get_config('PathStyle')) || 'staircase';
 }

 # where is our cache directory (if any)
 sub cache_dir     {
    my $self = shift;
-   return unless my $dir  = $self->r->dir_config('CacheDir');
+   return unless my $dir  = $self->get_config('CacheDir');
    return $self->r->server_root_relative($dir);
 }

 # columns to display
-sub subdir_columns {shift->r->dir_config('SubdirColumns') || SUBDIRCOLUMNS   }
-sub playlist_columns {shift->r->dir_config('PlaylistColumns') || PLAYLISTCOLUMNS }
+sub subdir_columns {shift->get_config('SubdirColumns') || SUBDIRCOLUMNS   }
+sub playlist_columns {shift->get_config('PlaylistColumns') || PLAYLISTCOLUMNS }

 # various configuration variables
-sub default_dir    { shift->r->dir_config('BaseDir') || BASE_DIR  }
+sub default_dir    { shift->get_config('BaseDir') || BASE_DIR  }
 sub stylesheet     { shift->get_dir('Stylesheet', STYLESHEET)     }
 sub parent_icon    { shift->get_dir('ParentIcon',PARENTICON)      }
 sub cd_list_icon   {
    my $self  = shift;
    my $subdir = shift;
-   my $image = $self->r->dir_config('CoverImageSmall') || COVERIMAGESMALL;
+   my $image = $self->get_config('CoverImageSmall') || COVERIMAGESMALL;
    my $directory_specific_icon = $self->r->filename."/$subdir/$image";
    return -e $directory_specific_icon
      ? join ("/",$self->r->uri,escape($subdir),$image)
@@ -1427,7 +1435,7 @@
 }
 sub playlist_icon {
    my $self = shift;
-   my $image = $self->r->dir_config('PlaylistImage') || PLAYLISTIMAGE;
+   my $image = $self->get_config('PlaylistImage') || PLAYLISTIMAGE;
    my $directory_specific_icon = $self->r->filename."/$image";
 warn $directory_specific_icon;
    return -e $directory_specific_icon
```

```
@@ -1444,7 +1452,7 @@
 sub cd_icon {
    my $self = shift;
    my $dir = shift;
-   my $coverimg = $self->r->dir_config('CoverImage') || COVERIMAGE;
+   my $coverimg = $self->get_config('CoverImage') || COVERIMAGE;
    if (-e "$dir/$coverimg") {
      $coverimg;
    } else {
@@ -1453,7 +1461,7 @@
 }
 sub missing_comment {
    my $self = shift;
-   my $missing = $self->r->dir_config('MissingComment');
+   my $missing = $self->get_config('MissingComment');
    return if $missing eq 'off';
    $missing = $self->lh->maketext('unknown') unless $missing;
    $missing;
@@ -1464,7 +1472,7 @@
    my $self = shift;
    my $data = shift;
    my $description;
-   my $format = $self->r->dir_config('DescriptionFormat');
+   my $format = $self->get_config('DescriptionFormat');
    if ($format) {
      ($description = $format) =~ s{%([atfglncrdmsqS%])}
        {$1 eq '%' ? '%'
@@ -1495,7 +1503,7 @@
      }
    }

-   if ((my $basename = $r->dir_config('StreamBase')) && !$self->is_localnet()) {
+   if ((my $basename = $self->get_config('StreamBase')) && !$self->is_localnet()) {
      $basename =~ s!http://!http://$auth_info! if $auth_info;
      return $basename;
    }
@@ -1536,7 +1544,7 @@
 sub is_localnet {
    my $self = shift;
    return 1 if $self->is_local;  # d'uh
-   my @local = split /\s+/,$self->r->dir_config('LocalNet') or return;
+   my @local = split /\s+/,$self->get_config('LocalNet') or return;

    my $remote_ip = $self->r->connection->remote_ip . '.';
    foreach (@local) {
```

, it was 194 lines long so I didn't inline it here, but it was quick to create with a few regexes search-n-replace manipulations in xemacs.

Now I have the browsing of the root */mp3/* directory and its sub-directories working. If I click on *'Fetch'* of a particular song it works too. However if I try to *'Stream'* a song, I get a 500 response with error_log telling me:

```
[Fri Jun 06 15:33:33 2003] [error] [client 127.0.0.1] Bad arg length
for Socket::unpack_sockaddr_in, length is 31, should be 16 at
.../5.9.0/i686-linux-thread-multi/Socket.pm line 370.
```

It would be certainly nice for *Socket.pm* to use `Carp::carp()` instead of `warn()` so we will know where in the `Apache::MP3` code this problem was triggered. However reading the *Socket.pm* manpage reveals that `sockaddr_in()` in the list context is the same as calling an explicit `unpack_sockaddr_in()`, and in the scalar context it's calling `pack_sockaddr_in()`. So I have found `sockaddr_in` was the only *Socket.pm* function used in `Apache::MP3` and I have found this code in the function `is_local()`:

```
my $r = $self->r;
my ($serverport,$serveraddr) = sockaddr_in($r->connection->local_addr);
my ($remoteport,$remoteaddr) = sockaddr_in($r->connection->remote_addr);
return $serveraddr eq $remoteaddr;
```

Since something is wrong with function calls `$r->connection->local_addr` and/or `$r->connection->remote_addr` and I referred to the mod_perl 1.0 backward compatibility document and found the relevant entry on these two functions. Indeed the API have changed. Instead of returning a packed SOCKADDR_IN string, Apache now returns an *APR::SocketAddr* object, which I can query to get the bits of information I'm interested in. So I applied this patch:

```
--- Apache/MP3.pm.3     2003-06-06 15:36:15.000000000 +1000
+++ Apache/MP3.pm       2003-06-06 15:56:32.000000000 +1000
@@ -1533,10 +1533,9 @@
 # allows the player to fast forward, pause, etc.
 sub is_local {
   my $self = shift;
-  my $r = $self->r;
-  my ($serverport,$serveraddr) = sockaddr_in($r->connection->local_addr);
-  my ($remoteport,$remoteaddr) = sockaddr_in($r->connection->remote_addr);
-  return $serveraddr eq $remoteaddr;
+  my $c = $self->r->connection;
+  require APR::SockAddr;
+  return $c->local_addr->ip_get eq $c->remote_addr->ip_get;
 }

 # Check if the requesting client is on the local network, as defined by
```

And voila, the streaming option now works. I get a warning on *'Use of uninitialized value'* on line 1516 though, but again this is unrelated to the porting issues, just a flow logic problem, which wasn't triggered without the warnings mode turned on. I have fixed it with:

```
--- Apache/MP3.pm.4     2003-06-06 15:57:15.000000000 +1000
+++ Apache/MP3.pm       2003-06-06 16:04:48.000
@@ -1492,7 +1492,7 @@
   my $suppress_auth = shift;
   my $r = $self->r;

-  my $auth_info;
+  my $auth_info = '';
   # the check for auth_name() prevents an anno
   # the apache server log when authentication
   if ($r->auth_name && !$suppress_auth) {
```

```
@@ -1509,10 +1509,9 @@
    }

   my $vhost = $r->hostname;
-  unless ($vhost) {
-     $vhost = $r->server->server_hostname;
-     $vhost .= ':' . $r->get_server_port unless
-  }
+  $vhost = $r->server->server_hostname unless
+  $vhost .= ':' . $r->get_server_port unless $
+
   return "http://${auth_info}${vhost}";
 }
```

This completes the first part of the porting. I have tried to use all the visible functions of the interface and everything seemed to work and I haven't got any warnings logged. Certainly I may have missed some usage patterns which may be still problematic. But this is good enough for this tutorial.

1.5.6.3 Getting Rid of the **Apache::compat** Dependency

The final stage is going to get rid of Apache::compat since this is a CPAN module, which must not load Apache::compat on its own. I'm going to make Apache::MP3 work with mod_perl 2.0 all by itself.

The first step is to comment out the loading of Apache::compat in *startup.pl*:

```
#file:startup.pl
#---------------
use Apache2 ();
use lib qw(/home/httpd/2.0/perl);
#use Apache::compat ();
```

1.5.6.4 Ensuring that **Apache::compat** is not loaded

The second step is to make sure that Apache::compat doesn't get loaded indirectly, through some other module. So I've added this line of code to *Apache/MP3.pm*:

```
--- Apache/MP3.pm.5     2003-06-06 16:17:50.000000000 +1000
+++ Apache/MP3.pm       2003-06-06 16:21:14.000000000 +1000
@@ -3,2 +3,6 @@

+BEGIN {
+    die "Apache::compat is loaded loaded" if $INC{'Apache/compat.pm'};
+}
+
 use strict;
```

and indeed, even though I've commented out the loading of Apache::compat from *startup.pl*, this module was still getting loaded. I knew that because the request to */mp3* were failing with the error message:

```
Apache::compat is loaded loaded at ...
```

There are several ways to find the guilty party, you can grep(1) for it in the perl libraries, you can override CORE::GLOBAL::require() in *startup.pl*:

```
BEGIN {
  use Carp;
  *CORE::GLOBAL::require = sub {
      Carp::cluck("Apache::compat is loaded") if $_[0] =~ /compat/;
      CORE::require(@_);
  };
}
```

or you can modify *Apache/compat.pm* and make it print the calls trace when it gets compiled:

```
--- Apache/compat.pm.orig   2003-06-03 16:11:07.000000000 +1000
+++ Apache/compat.pm        2003-06-03 16:11:58.000000000 +1000
@@ -1,5 +1,9 @@
 package Apache::compat;

+BEGIN {
+    use Carp;
+    Carp::cluck("Apache::compat is loaded by");
+}
```

I've used this last technique, since it's the safest one to use. Remember that Apache::compat can also be loaded with:

```
    do "Apache/compat.pm";
```

in which case, neither grep(1)'ping for Apache::compat, nor overriding require() will do the job.

When I've restarted the server and tried to use Apache::MP3 (I wasn't preloading it at the server startup since I wanted the server to start normally and cope with problem when it's running), the *error_log* had an entry:

```
Apache::compat is loaded by at .../Apache2/Apache/compat.pm line 6
  Apache::compat::BEGIN() called at .../Apache2/Apache/compat.pm line 8
  eval {...} called at .../Apache2/Apache/compat.pm line 8
  require Apache/compat.pm called at .../5.9.0/CGI.pm line 169
  require CGI.pm called at .../site_perl/5.9.0/Apache/MP3.pm line 8
  Apache::MP3::BEGIN() called at .../Apache2/Apache/compat.pm line 8
```

(I've trimmed the whole paths of the libraries and the trace itself, to make it easier to understand.)

We could have used Carp::carp() which would have told us only the fact that Apache::compat was loaded by CGI.pm, but by using Carp::cluck() we've obtained the whole stack backtrace so we also can learn which module has loaded CGI.pm.

Here I've learned that I had an old version of CGI.pm (2.89) which automatically loaded Apache::compat (which should be never done by CPAN modules). Once I've upgraded CGI.pm to version 2.93 and restarted the server, Apache::compat wasn't getting loaded any longer.

1.5.6.5 Installing the `ModPerl::MethodLookup` Helper

Now that `Apache::compat` is not loaded, I need to deal with two issues: modules that need to be loaded and APIs that have changed.

For the second issue I'll have to refer to the the mod_perl 1.0 backward compatibility document.

But the first issue can be easily worked out using `ModPerl::MethodLookup`. As explained in the section Using `ModPerl::MethodLookup` Programmatically I've added the `AUTOLOAD` code to my *startup.pl* so it'll automatically lookup the packages that I need to load based on the request method and the object type.

So now my *startup.pl* looked like:

```
#file:startup.pl
#---------------
use Apache2 ();
use lib qw(/home/httpd/2.0/perl);

{
  package ModPerl::MethodLookupAuto;
  use ModPerl::MethodLookup;

  use Carp;
  sub handler {

      # look inside mod_perl:: Apache:: APR:: ModPerl:: excluding DESTROY
      my $skip = '^(?!DESTROY$';
      *UNIVERSAL::AUTOLOAD = sub {
          my $method = $AUTOLOAD;
          return if $method =~ /DESTROY/;
          my ($hint, @modules) =
              ModPerl::MethodLookup::lookup_method($method, @_);
          $hint ||= "Can't find method $AUTOLOAD";
          croak $hint;
      };
      return 0;
  }
}
1;
```

and I add to my *httpd.conf*:

```
PerlChildInitHandler ModPerl::MethodLookupAuto
```

1.5.6.6 Adjusting the code to run under mod_perl 2

I restart the server and off I go to complete the second porting stage.

The first error that I've received was:

```
[Fri Jun 06 16:28:32 2003] [error] failed to resolve handler 'Apache::MP3'
[Fri Jun 06 16:28:32 2003] [error] [client 127.0.0.1] Can't locate
object method "boot" via package "mod_perl" at .../Apache/Constants.pm
line 8. Compilation failed in require at .../Apache/MP3.pm line 12.
```

I go to line 12 and find the following code:

```
use Apache::Constants qw(:common REDIRECT HTTP_NO_CONTENT
                         DIR_MAGIC_TYPE HTTP_NOT_MODIFIED);
```

Notice that I did have mod_perl 1.0 installed, so the `Apache::Constant` module from mod_perl 1.0 couldn't find the `boot()` method which doesn't exist in mod_perl 2.0. If you don't have mod_perl 1.0 installed the error would simply say, that it can't find *Apache/Constants.pm* in `@INC`. In any case, we are going to replace this code with mod_perl 2.0 equivalent:

```
--- Apache/MP3.pm.6     2003-06-06 16:33:05.000000000 +1000
+++ Apache/MP3.pm       2003-06-06 17:03:43.000000000 +1000
@@ -9,7 +9,9 @@
 use warnings;
 no warnings 'redefine';  # XXX: remove when done with porting

-use Apache::Constants qw(:common REDIRECT HTTP_NO_CONTENT DIR_MAGIC_TYPE HTTP_NOT_MODIFIED);
+use Apache::Const -compile => qw(:common REDIRECT HTTP_NO_CONTENT
+                                 DIR_MAGIC_TYPE HTTP_NOT_MODIFIED);
+
 use Apache::MP3::L10N;
 use IO::File;
 use Socket 'sockaddr_in';
```

and I also had to adjust the constants, since what used to be `OK`, now has to be `Apache::OK`, mainly because in mod_perl 2.0 there is an enormous amount of constants (coming from Apache and APR) and most of them are grouped in `Apache::` or `APR::` namespaces. The `Apache::Const` and `APR::Const` manpage provide more information on available constants.

This search and replace accomplished the job:

```
% perl -pi -e 's/return\s(OK|DECLINED|FORBIDDEN| \
  REDIRECT|HTTP_NO_CONTENT|DIR_MAGIC_TYPE| \
  HTTP_NOT_MODIFIED)/return Apache::$1/xg' Apache/MP3.pm
```

As you can see the regex explicitly lists all constants that were used in `Apache::MP3`. Your situation may vary. Here is the patch: *code/apache_mp3_7.diff*:

```
--- Apache/MP3.pm.7     2003-06-06 17:04:27.000000000 +1000
+++ Apache/MP3.pm       2003-06-06 17:13:26.000000000 +1000
@@ -129,7 +129,7 @@
   my $self = shift;

   $self->r->send_http_header( $self->html_content_type );
-  return OK if $self->r->header_only;
+  return Apache::OK if $self->r->header_only;

   print start_html(
                   -lang => $self->lh->language_tag,
@@ -246,20 +246,20 @@
       $self->send_playlist(\@matches);
```

```
      }

-     return OK;
+     return Apache::OK;
    }

    # this is called to generate a playlist for selected files
    if (param('Play Selected')) {
-     return HTTP_NO_CONTENT unless my @files = param('file');
+     return Apache::HTTP_NO_CONTENT unless my @files = param('file');
      my $uri = dirname($r->uri);
      $self->send_playlist([map { "$uri/$_" } @files]);
-     return OK;
+     return Apache::OK;
    }

    # otherwise don't know how to deal with this
    $self->r->log_reason('Invalid parameters -- possible attempt to circumvent checks.');
-   return FORBIDDEN;
+   return Apache::FORBIDDEN;
 }

 # this generates the top-level directory listing
@@ -273,7 +273,7 @@
    my $query = $self->r->args;
    $query = "?" . $query if defined $query;
    $self->r->header_out(Location => "$uri/$query");
-     return REDIRECT;
+     return Apache::REDIRECT;
    }

   return $self->list_directory($dir);
@@ -289,9 +289,9 @@

   if ($is_audio && !$self->download_ok) {
     $self->r->log_reason('File downloading is forbidden');
-     return FORBIDDEN;
+     return Apache::FORBIDDEN;
   } else {
-     return DECLINED;  # allow Apache to do its standard thing
+     return Apache::DECLINED;  # allow Apache to do its standard thing
   }

 }
@@ -302,17 +302,17 @@
   my $self = shift;
   my $r = $self->r;

-   return DECLINED unless -e $r->filename;  # should be $r->finfo
+   return Apache::DECLINED unless -e $r->filename;  # should be $r->finfo

   unless ($self->stream_ok) {
     $r->log_reason('AllowStream forbidden');
-     return FORBIDDEN;
+     return Apache::FORBIDDEN;
   }

   if ($self->check_stream_client and !$self->is_stream_client) {
     my $useragent = $r->header_in('User-Agent');
```

```
      $r->log_reason("CheckStreamClient is true and $useragent is not a streaming client");
-     return FORBIDDEN;
+     return Apache::FORBIDDEN;
    }

    return $self->send_stream($r->filename,$r->uri);
@@ -322,12 +322,12 @@
 sub send_playlist {
    my $self = shift;
    my ($urls,$shuffle) = @_;
-   return HTTP_NO_CONTENT unless @$urls;
+   return Apache::HTTP_NO_CONTENT unless @$urls;
    my $r = $self->r;
    my $base = $self->stream_base;

    $r->send_http_header('audio/mpegurl');
-   return OK if $r->header_only;
+   return Apache::OK if $r->header_only;

    # local user
    my $local = $self->playlocal_ok && $self->is_local;
@@ -377,7 +377,7 @@
        $r->print ("$base$_?$stream_parms$CRLF");
      }
    }
-   return OK;
+   return Apache::OK;
 }

 sub stream_parms {
@@ -468,7 +468,7 @@
    my $self = shift;
    my $dir  = shift;

-   return DECLINED unless -d $dir;
+   return Apache::DECLINED unless -d $dir;

    my $last_modified = (stat(_))[9];

@@ -478,15 +478,15 @@
     my ($time, $ver) = $check =~ /^([a-f0-9]+)-([0-9.]+)$/;

     if ($check eq '*' or (hex($time) == $last_modified and $ver == $VERSION)) {
-      return HTTP_NOT_MODIFIED;
+      return Apache::HTTP_NOT_MODIFIED;
     }
    }

-   return DECLINED unless my ($directories,$mp3s,$playlists,$txtfiles)
+   return Apache::DECLINED unless my ($directories,$mp3s,$playlists,$txtfiles)
     = $self->read_directory($dir);

    $self->r->send_http_header( $self->html_content_type );
-   return OK if $self->r->header_only;
+   return Apache::OK if $self->r->header_only;

    $self->page_top($dir);
    $self->directory_top($dir);
@@ -514,7 +514,7 @@
```

```
    print hr                           unless %$mp3s;
    print "\n\n";
    $self->directory_bottom($dir);
-   return OK;
+   return Apache::OK;
 }

 # print the HTML at the top of the page
@@ -1268,8 +1268,8 @@

    my $mime = $r->content_type;
    my $info = $self->fetch_info($file,$mime);
-   return DECLINED unless $info;  # not a legit mp3 file?
-   my $fh = $self->open_file($file) || return DECLINED;
+   return Apache::DECLINED unless $info;  # not a legit mp3 file?
+   my $fh = $self->open_file($file) || return Apache::DECLINED;
    binmode($fh);  # to prevent DOS text-mode foolishness

    my $size = -s $file;
@@ -1317,7 +1317,7 @@
    $r->print("Content-Length: $size$CRLF");
    $r->print("Content-Type: $mime$CRLF");
    $r->print("$CRLF");
-   return OK if $r->header_only;
+   return Apache::OK if $r->header_only;

    if (my $timeout = $self->stream_timeout) {
      my $seconds  = $info->{seconds};
@@ -1330,12 +1330,12 @@
        $bytes -= $b;
        $r->print($data);
      }
-     return OK;
+     return Apache::OK;
    }

    # we get here for untimed transmits
    $r->send_fd($fh);
-   return OK;
+   return Apache::OK;
 }

 # called to open the MP3 file
```

.

I had to manually fix the `DIR_MAGIC_TYPE` constant which didn't fit the regex pattern:

```
  --- Apache/MP3.pm.8     2003-06-06 17:24:33.000000000 +1000
  +++ Apache/MP3.pm       2003-06-06 17:26:29.000000000 +1000
  @@ -1055,7 +1055,7 @@

      my $mime = $self->r->lookup_file("$dir/$d")->content_type;

  -     push(@directories,$d) if !$seen{$d}++ && $mime eq DIR_MAGIC_TYPE;
  +     push(@directories,$d) if !$seen{$d}++ && $mime eq Apache::DIR_MAGIC_TYPE;

      # .m3u files should be configured as audio/playlist MIME types in your apache .conf file
      push(@playlists,$d) if $mime =~ m!^audio/(playlist|x-mpegurl|mpegurl|x-scpls)$!;
```

And I move on, the next error is:

```
[Fri Jun 06 17:28:00 2003] [error] [client 127.0.0.1]
Can't locate object method "header_in" via package
"Apache::RequestRec" at .../Apache/MP3.pm line 85.
```

The porting document quickly reveals me that `header_in()` and its brothers `header_out()` and `err_header_out()` are R.I.P. and that I have to use the corresponding functions `headers_in()`, `headers_out()` and `err_headers_out()` which are available in mod_perl 1.0 API as well.

So I adjust the code to use the new API:

```
% perl -pi -e 's|header_in\((.*?)\)|headers_in->{$1}|g' Apache/MP3.pm
% perl -pi -e 's|header_out\((.*?)\s*=>\s*(.*?)\);|headers_out->{$1} = $2;|g' Apache/MP3.pm
```

which results in this patch: *code/apache_mp3_9.diff*:

```
--- Apache/MP3.pm.9     2003-06-06 17:27:45.000000000 +1000
+++ Apache/MP3.pm       2003-06-06 17:55:14.000000000 +1000
@@ -82,8 +82,8 @@
   $new->{'r'} ||= $r if $r;

   my @lang_tags;
-  push @lang_tags,split /,\s+/,$r->header_in('Accept-language')
-    if $r->header_in('Accept-language');
+  push @lang_tags,split /,\s+/,$r->headers_in->{'Accept-language'}
+    if $r->headers_in->{'Accept-language'};
   push @lang_tags,$new->get_config('DefaultLanguage') || 'en-US';

   $new->{'lh'} ||=
@@ -272,7 +272,7 @@
    my $uri = $self->r->uri;
    my $query = $self->r->args;
    $query = "?" . $query if defined $query;
-   $self->r->header_out(Location => "$uri/$query");
+   $self->r->headers_out->{Location} = "$uri/$query";
    return Apache::REDIRECT;
   }

@@ -310,7 +310,7 @@
   }

   if ($self->check_stream_client and !$self->is_stream_client) {
-    my $useragent = $r->header_in('User-Agent');
+    my $useragent = $r->headers_in->{'User-Agent'};
     $r->log_reason("CheckStreamClient is true and $useragent is not a streaming client");
     return Apache::FORBIDDEN;
   }
@@ -472,9 +472,9 @@

   my $last_modified = (stat(_))[9];

-  $self->r->header_out('ETag' => sprintf("%lx-%s", $last_modified, $VERSION));
+  $self->r->headers_out->{'ETag'} = sprintf("%lx-%s", $last_modified, $VERSION);

-  if (my $check = $self->r->header_in("If-None-Match")) {
+  if (my $check = $self->r->headers_in->{"If-None-Match"}) {
```

```
     my ($time, $ver) = $check =~ /^([a-f0-9]+)-([0-9.]+)$/;

     if ($check eq '*' or (hex($time) == $last_modified and $ver == $VERSION)) {
@@ -1283,8 +1283,8 @@
   my $genre       = $info->{genre} || $self->lh->maketext('unknown');

   my $range = 0;
- $r->header_in("Range")
-   and $r->header_in("Range") =~ m/bytes=(\d+)/
+ $r->headers_in->{"Range"}
+   and $r->headers_in->{"Range"} =~ m/bytes=(\d+)/
     and $range = $1
     and seek($fh,$range,0);

@@ -1383,11 +1383,11 @@
 # return true if client can stream
 sub is_stream_client {
   my $r = shift->r;
- $r->header_in('Icy-MetaData')   # winamp/xmms
-   || $r->header_in('Bandwidth')   # realplayer
-     || $r->header_in('Accept') =~ m!\baudio/mpeg\b!  # mpg123 and others
-       || $r->header_in('User-Agent') =~ m!^NSPlayer/!  # Microsoft media player
-         || $r->header_in('User-Agent') =~ m!^xmms/!;
+ $r->headers_in->{'Icy-MetaData'}   # winamp/xmms
+   || $r->headers_in->{'Bandwidth'}   # realplayer
+     || $r->headers_in->{'Accept'} =~ m!\baudio/mpeg\b!  # mpg123 and others
+       || $r->headers_in->{'User-Agent'} =~ m!^NSPlayer/!  # Microsoft media player
+         || $r->headers_in->{'User-Agent'} =~ m!^xmms/!;
 }

 # whether to read info for each MP3 file (might take a long time)
```

.

On the next error `ModPerl::MethodLookup`'s `AUTOLOAD` kicks in. Instead of complaining:

```
[Fri Jun 06 18:35:53 2003] [error] [client 127.0.0.1]
Can't locate object method "FETCH" via package "APR::Table"
at .../Apache/MP3.pm line 85.
```

I now get:

```
[Fri Jun 06 18:36:35 2003] [error] [client 127.0.0.1]
to use method 'FETCH' add:
     use APR::Table ();
at .../Apache/MP3.pm line 85
```

So I follow the suggestion and load `APR::Table()`:

```
--- Apache/MP3.pm.10     2003-06-06 17:57:54.000000000 +1000
+++ Apache/MP3.pm        2003-06-06 18:37:33.000000000 +1000
@@ -9,6 +9,8 @@
 use warnings;
 no warnings 'redefine';  # XXX: remove when done with porting

+use APR::Table ();
+
 use Apache::Const -compile => qw(:common REDIRECT HTTP_NO_CONTENT
                                  DIR_MAGIC_TYPE HTTP_NOT_MODIFIED);
```

I continue issuing the request and adding the missing modules again and again till I get no more
complaints. During this process I've added the following modules:

```
--- Apache/MP3.pm.11     2003-06-06 18:38:47.000000000 +1000
+++ Apache/MP3.pm        2003-06-06 18:39:10.000000000 +1000
@@ -9,6 +9,14 @@
 use warnings;
 no warnings 'redefine';  # XXX: remove when done with porting

+use Apache::Connection ();
+use Apache::SubRequest ();
+use Apache::Access ();
+use Apache::RequestIO ();
+use Apache::RequestUtil ();
+use Apache::RequestRec ();
+use Apache::ServerUtil ();
+use Apache::Log;
 use APR::Table ();

 use Apache::Const -compile => qw(:common REDIRECT HTTP_NO_CONTENT
```

The AUTOLOAD code helped me to trace the modules that contain the existing APIs, however I still have
to deal with APIs that no longer exist. Rightfully the helper code says that it doesn't know which module
defines the method: `send_http_header()` because it no longer exists in Apache 2.0 vocabulary:

```
[Fri Jun 06 18:40:34 2003] [error] [client 127.0.0.1]
Don't know anything about method 'send_http_header'
at .../Apache/MP3.pm line 498
```

So I go back to the porting document and find the relevant entry. In 2.0 lingo, we just need to set the
`content_type()`:

```
--- Apache/MP3.pm.12     2003-06-06 18:43:42.000000000 +1000
+++ Apache/MP3.pm        2003-06-06 18:51:23.000000000 +1000
@@ -138,7 +138,7 @@
 sub help_screen {
   my $self = shift;

-  $self->r->send_http_header( $self->html_content_type );
+  $self->r->content_type( $self->html_content_type );
   return Apache::OK if $self->r->header_only;

   print start_html(
@@ -336,7 +336,7 @@
```

```
   my $r = $self->r;
   my $base = $self->stream_base;

-  $r->send_http_header('audio/mpegurl');
+  $r->content_type('audio/mpegurl');
   return Apache::OK if $r->header_only;

   # local user
@@ -495,7 +495,7 @@
   return Apache::DECLINED unless my ($directories,$mp3s,$playlists,$txtfiles)
     = $self->read_directory($dir);

-  $self->r->send_http_header( $self->html_content_type );
+  $self->r->content_type( $self->html_content_type );
   return Apache::OK if $self->r->header_only;

   $self->page_top($dir);
```

also I've noticed that there was this code:

```
   return Apache::OK if $self->r->header_only;
```

This technique is no longer needed in 2.0, since Apache 2.0 automatically discards the body if the request is of type `HEAD` -- the handler should still deliver the whole body, which helps to calculate the content-length if this is relevant to play nicer with proxies. So you may decide not to make a special case for `HEAD` requests.

At this point I was able to browse the directories and play files via most options without relying on `Apache::compat`.

There were a few other APIs that I had to fix in the same way, while trying to use the application, looking at the *error_log* referring to the porting document and applying the suggested fixes. I'll make sure to send all these fixes to Lincoln Stein, so the new versions will work correctly with mod_perl 2.0. I also had to fix other `Apache::MP3::` files, which come as a part of the `Apache-MP3` distribution, pretty much using the same techniques explained here. A few extra fixes of interest in `Apache::MP3` were:

- **send_fd()**

  As of this writing we don't have this function in the core, because Apache 2.0 doesn't have it (it's in `Apache::compat` but implemented in a slow way). However we may provide one in the future. Currently one can use the function `sendfile()` which requires a filename as an argument and not the file descriptor. So I have fixed the code:

  ```
  -      if($r->request($r->uri)->content_type eq 'audio/x-scpls'){
  -        open(FILE,$r->filename) || return 404;
  -        $r->send_fd(\*FILE);
  -        close(FILE);
  +
  +      if($r->content_type eq 'audio/x-scpls'){
  +        $r->sendfile($r->filename) || return Apache::NOT_FOUND;
  ```

- **log_reason**

`log_reason` is now `log_error`:

```
    -  $self->r->log_reason('Invalid parameters -- possible attempt to circumvent checks.');
    +  $r->log_error('Invalid parameters -- possible attempt to circumvent checks.')
    ;
```

I have found the porting process to be quite interesting, especially since I have found several bugs in Apache 2.0 and documented a few undocumented API changes. It was also fun, because I've got to listen to mp3 files when I did things right, and was getting silence in my headphones and a visual irritation in the form of *error_log* messages when I didn't ;)

1.6 Porting a Module to Run under both mod_perl 2.0 and mod_perl 1.0

Sometimes code needs to work with both mod_perl versions. For example this is the case with CPAN module developers who wish to continue to maintain a single code base, rather than supplying two separate implementations.

1.6.1 Making Code Conditional on Running mod_perl Version

In this case you can test for which version of mod_perl your code is running under and act appropriately.

To continue our example above, let's say we want to support opening a filehandle in both mod_perl 2.0 and mod_perl 1.0. Our code can make use of the variable `$mod_perl::VERSION`:

```
use mod_perl;
use constant MP2 => ($mod_perl::VERSION >= 1.99);
# ...
require Symbol if MP2;
# ...

my $fh = MP2 ? Symbol::gensym : Apache->gensym;
open $fh, $file or die "Can't open $file: $!";
```

Though, make sure that you don't use `$mod_perl::VERSION` string anywhere in the code before you have declared your module's own `$VERSION`, since PAUSE will pick the wrong version when you submit the module on CPAN. It requires that module's `$VERSION` will be declared first. You can verify whether it'll pick the *Foo.pm*'s version correctly, by running this code:

```
% perl -MExtUtils::MakeMaker -le 'print MM->parse_version(shift)' Foo.pm
```

There is more information about this issue here:
http://pause.perl.org/pause/query?ACTION=pause_04about#conventions

Some modules, like `CGI.pm` may work under mod_perl and without it, and will want to use the mod_perl 1.0 API if that's available, or mod_perl 2.0 API otherwise. So the following idiom could be used for this purpose.

```
use constant MP_GEN => $ENV{MOD_PERL}
    ? eval { require mod_perl;  $mod_perl::VERSION >= 1.99 ? 2 : 1 }
    : 0;
```

It sets the constant `MP_GEN` to 0 if mod_perl is not available, to 1 if running under mod_perl 1.0 and 2 for mod_perl 2.0.

Here's another way to find out the mod_perl version. In the server configuration file you can use a special configuration "define" symbol `MODPERL2`, which is magically enabled internally, as if the server had been started with `-DMODPERL2`.

```
# in httpd.conf
<IfDefine MODPERL2>
    # 2.0 configuration
</IfDefine>
<IfDefine !MODPERL2>
    # else
</IfDefine>
```

From within Perl code this can be tested with `Apache::Server::exists_config_define()`. For example, we can use this method to decide whether or not to call `$r->send_http_header()`, which no longer exists in mod_perl 2.0:

```
sub handler {
    my $r = shift;
    $r->content_type('text/html');
    $r->send_http_header() unless Apache::Server::exists_config_define("MODPERL2");
    ...
}
```

Relevant links to other places in the porting documents:

- *mod_perl 1.0 and 2.0 Constants Coexistence*

1.6.2 Method Handlers

Method handlers in mod_perl are declared using the *'method'* attribute. However if you want to have the same code base for mod_perl 1.0 and 2.0 applications, whose handler has to be a method, you will need to do the following trick:

```
sub handler_mp1 ($$)     { ... }
sub handler_mp2 : method { ... }
*handler = MP2 ? \&handler_mp2 : \&handler_mp1;
```

Note that this requires at least Perl 5.6.0, the *:method* attribute is not supported by older Perl versions, which will fail to compile such code.

Here are two complete examples. The first example implements `MyApache::Method` which has a single method that works for both mod_perl generations:

The configuration:

```
PerlModule MyApache::Method
<Location /method>
    SetHandler perl-script
    PerlHandler MyApache::Method->handler
</Location>
```

The code:

```
#file:MyApache/Method.pm
package MyApache::Method;

# PerlModule MyApache::Method
# <Location /method>
#      SetHandler perl-script
#      PerlHandler MyApache::Method->handler
#  </Location>

use strict;
use warnings;

use mod_perl;
use constant MP2 => $mod_perl::VERSION < 1.99 ? 0 : 1;

BEGIN {
    if (MP2) {
        require Apache::RequestRec;
        require Apache::RequestIO;
        require Apache::Const;
        Apache::Const->import(-compile => 'OK');
    }
    else {
        require Apache;
        require Apache::Constants;
        Apache::Constants->import('OK');
    }
}

sub handler_mp1 ($$)      { &run }
sub handler_mp2 : method { &run }
*handler = MP2 ? \&handler_mp2 : \&handler_mp1;

sub run {
    my($class, $r) = @_;
    MP2 ? $r->content_type('text/plain')
        : $r->send_http_header('text/plain');
    print "$class was called\n";
    return MP2 ? Apache::OK : Apache::Constants::OK;
}
```

Here are two complete examples. The second example implements MyApache::Method2, which is very similar to MyApache::Method, but uses separate methods for mod_perl 1.0 and 2.0 servers.

The configuration is the same:

```
PerlModule MyApache::Method2
<Location /method2>
    SetHandler perl-script
    PerlHandler MyApache::Method2->handler
</Location>
```

The code:

```
#file:MyApache/Method2.pm
package MyApache::Method2;

# PerlModule MyApache::Method
# <Location /method>
#       SetHandler perl-script
#       PerlHandler MyApache::Method->handler
#  </Location>

use strict;
use warnings;

use mod_perl;
use constant MP2 => $mod_perl::VERSION < 1.99 ? 0 : 1;

BEGIN {
    warn "running $mod_perl::VERSION!\n";
    if (MP2) {
        require Apache::RequestRec;
        require Apache::RequestIO;
        require Apache::Const;
        Apache::Const->import(-compile => 'OK');
    }
    else {
        require Apache;
        require Apache::Constants;
        Apache::Constants->import('OK');
    }
}

sub handler_mp1 ($$)      { &mp1 }
sub handler_mp2 : method { &mp2 }

*handler = MP2 ? \&handler_mp2 : \&handler_mp1;

sub mp1 {
   my($class, $r) = @_;
   $r->send_http_header('text/plain');
   $r->print("mp1: $class was called\n");
   return Apache::Constants::OK();
```

```
    }

    sub mp2 {
        my($class, $r) = @_;
        $r->content_type('text/plain');
        $r->print("mp2: $class was called\n");
        return Apache::OK();
    }
```

Assuming that mod_perl 1.0 is listening on port 8001 and mod_perl 2.0 on 8002, we get the following results:

```
% lynx --source http://localhost:8001/method
MyApache::Method was called

% lynx --source http://localhost:8001/method2
mp1: MyApache::Method2 was called

% lynx --source http://localhost:8002/method
MyApache::Method was called

% lynx --source http://localhost:8002/method2
mp2: MyApache::Method2 was called
```

1.7 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

Stas Bekman <stas (at) stason.org>

1.8 Authors

- Nick Tonkin <nick (at) tonkinresolutions.com>

- Stas Bekman <stas (at) stason.org>

Only the major authors are listed above. For contributors see the Changes file.

Table of Contents: