1 Code Snippets

1.1 Description

A collection of mod_perl code snippets which you can either adapt to your own use or integrate directly into your own code.

1.2 File Upload with Apache::Request

The `Apache::Request` module gives you an easy way to get form content, including uploaded files. In order to add file upload functionality to your form, you need to add two things.

First, you'll need to add a form field which is type *file*. This will put a `browse` button on the form that will allow the user to choose a file to upload.

Second, you'll neet to make sure to add, to the `form` tag the following:

```
enctype="multipart/form-data"
```

You won't be able to upload a file unless you have added this to the `form` tag.

In your code, you'll need to take a few extra steps to actually retrieve that file that has been uploaded. Using the following `form()` method will allow you to have a standard function that handles all of your forms, and does the right thing in the event that there was a file uploaded. You can put this function in your `mod_perl` handler, or in whatever module you want.

```
sub form {
    use Apache::Request;
    my $r = Apache->request();
    my $apr = Apache::Request->new($r);
    my @keys = $apr->param;

    my %form;
    foreach my $key(@keys) {

        my @value = $apr->param($key);
        next unless scalar @value;

        if ( @value > 1 ) {
            $form{$key} = \@value;
          } else {
            $form{$key} = $value[0];
        }
    }

    my $upload = $apr->upload;
    if ($upload) {
        $form{UPLOAD} = $upload;
    }

   return \%form;
}
```

In your code, you can get the contents of the form by calling this function:

```
my $form = Your::Class::form(); # Wherever you put this function
```

The value returned from this function is compatible with `CGI.pm` and other modules such as `CGI::Lite` Which is to say, the function returns a hashref. The keys of the hash are the names in your form. The values in the hash are the values entered in those fields, with the exception that a multiple select list with multiple things selected will return a listref of the selected values.

If your form contained a file upload element, then `$form{UPLOAD}` will contain a file upload object, which you can make calls back into.

For example:

```
my $form = Your::Class::form(); # Wherever you put this function
if (my $file = $form->{UPLOAD}) {
    my $filename = $file->filename; # If you need the name

    # And, if you want to save the file at $filelocation ...
    open F, ">$filelocation";
    my $filehandle = $file->fh;
    while (my $d = <$filehandle>) {
        print F $d;
    }
    close F;
}
```

That should give you the general idea of how this works. This lets you have a generic form handler that does "normal" forms as well as file upload forms, in mod_perl, without having to mess with `CGI.pm`, and without having to do custom things when you have a file upload.

You will need to see the documentation for `Apache::Upload` for more information about how to deal with the file upload object once you have it. Note that the `Apache::Upload` docs are embeded in the `Apache::Request` documentation, so you'll need to look there for that information.

1.3 Redirecting Errors to the Client Instead of error_log

Many error conditions result in an *exception* (or *signal* -- same thing) which is *raised* in order to tell the operating system that a condition has arisen which needs more urgent attention than can be given by other means. One of the most familiar ways of raising a signal is to hit `Ctrl-C` on your terminal's keyboard. The signal interrupts the processor. In the case of `Ctrl-C` the `INT` signal is generated and the interrupt is usually *trapped* by a default *signal handler* supplied by OS, which causes the operating system to stop the process currently attached to the terminal.

Under mod_perl, a Perl runtime error causes an exception. By default this exception is trapped by default mod_perl handler. The handler logs information about the error (such as the date and time that the error occurred) to *error_log*. If you want to redirect this information to the client instead of to *error_log* you have to take the responsibility yourself, by writing your own exception handler to implement this behaviour. See the section "Exception Handling for mod_perl" for more information.

The code examples below can be useful with your own exception handlers as well as with the default handlers.

META: Integrate the 2 sections

The CGI::Carp package implements handlers for signals. To trap (almost) all Perl run-time errors and send the output to the client instead of to Apache's `error_log` add this line to your script:

```
use CGI::Carp qw(fatalsToBrowser);
```

Refer to the `CGI::Carp` man page for more detailed information.

You can trap individual exceptions: for example you can write custom `__DIE__` and `__WARN__` signal handlers. The special `%SIG` hash contains references to signal handlers. The signal handler is just a subroutine, in the example below it is called "mydie". To install the handler we assign a reference to our handler to the appropriate element of the `%SIG` hash. This causes the signal handler to call mydie(error_message) whenever the `die()` sub is called as a result of something which happened when our script was executing.

Do not forget the `local` keyword! If you do, then after the signal handler has been loaded it will be called whenever `die()` is called by *any* script executed by the same process. Probably that's not what you want. If it is, you can put the assignment statement in any module, as long as it will be executed at the right time.

Here is an example of a handler which I wrote because I wanted users to know that there was an error, without displaying the error message, but instead email it to me. If the error is caused by user (e.g. uploading image whose size is bigger than the limit I had set) I want to tell them about it. I wrote this handler for the mod_perl environment, but it works correctly when called from the shell. The code shown below is a stripped-down version with additional comments.

The following code must be added to the script:

```
# Using the local() keyword restricts the scope of the directive to
# the block in which it is found, so this line must be added at the
# right place in the right script.  It will not affect other blocks
# unless the local() keyword is removed.  Usually you will want the
# directive to affect the entire script, so you just place it near
# the beginning of the file, where the innermost enclosing block is
# the file itself.
local $SIG{__DIE__} = \&mydie;

# The line above assumes that the subroutine "mydie" is in the same script.
# Alternatively you can create a separate module for the error handler.
# If you put the signal handler in a separate module, e.g. Error.pm,
# you must explicitly give the package name to set the handler in your
# script, using a line like this instead of the one above:
local $SIG{__DIE__} = \&Error::mydie;
# again within the script!

# Do not forget the C<local()>, unless you want this signal handler to
# be invoked every time any scripts dies (including events where this
# treatment may be undesirable).
```

```
my $max_image_size = 100*1024; # 100k
my $admin_email    = 'foo@example.com';

# and the handler itself
# Here is the handler itself:
# The handler is called with a text message in a scalar argument
sub mydie{
  my $why = shift;

  chomp $why;
  my $orig_why = $why;        # an ASCII copy for email report

  # handle the shell execution case (so we will not get all the HTML)
  print("Error: $why\n"), exit unless $ENV{MOD_PERL};

  my $should_email = 0;
  my $message = '';

  $why =~ s/[<&>]/"&#".ord($&).";"/ge;    # entity escape

  # Now we need to trap various kinds of errors that come from CGI.pm
  # We don't want these errors to be emailed to us, since
  # they aren't programmatical errors
  if ($orig_why =~ /Client attempted to POST (\d+) bytes/o) {

    $message = qq{
        You cannot POST messages bigger than
        @{[1024*$max_image_size]} bytes.<BR>
        You have tried to post $1 bytes<BR>
        If you are trying to upload an image, make sure its
        size is no bigger than @{[1024*$max_image_size]}
        bytes.<P>
        Thank you!
       };

  } elsif ($orig_why =~ /Malformed multipart POST/o) {

    $message = qq{
        Have you tried to upload an image in the wrong way?<P>
        To successfully upload an image you must use a browser that supports
        image upload and use the 'Browse' button to select that image.
        DO NOT type the path to the image into the upload field.<P>
        Thank you!

       };

  } elsif ($orig_why =~ /closed socket during multipart read/o) {

    $message = qq{
        Have you pressed a 'STOP' button?<BR>
        Please try again!<P>
        Thank you!
       };
```

```
    } else {

      $message = qq{
          <B>You need take no action since
          the error report has already been
          sent to the webmaster. <BR><P>
          <B>Thank you for your patience!</B>
         };

      $should_email = 1;
    }


    print qq{Content-type: text/html

  <HTML><BODY BGCOLOR="white">
  <B>Oops, Something went wrong.</B><P>
  $message
  </BODY></HTML>};

      # send email report if appropriate
    if ($should_email){

        # import sendmail subs
      use Mail ();
        # prepare the email error report:
      my $subject ="Error Report";
      my $body = qq|
    An error has happened:

    $orig_why

      |;

        # send error reports to admin
      send_mail($admin_email,$admin_email,$subject,$body);
      print STDERR "[".scalar localtime()."] [SIGDIE] Sending Error Email\n";
    }

       # print to error_log so we will know there was an error
    print STDERR "[".scalar localtime()."] [SIGDIE] $orig_why \n";

    exit 1;
  }             # end of sub mydie
```

You may have noticed that I trap the CGI.pm's die() calls here, I don't see any reason why my users should see ugly error messages, but that's the way CGI.pm written. The workaround is to trap them yourself.

Please note that as of version 2.49, CGI.pm provides the cgi_error() method to print the errors and won't die() unless you want it to.

1.4 Reusing Data from POST request

What happens if you need to access the POSTed data more than once, say to reuse it in subsequent handlers of the same request? POSTed data comes directly from the socket, and at the low level data can only be read from a socket once. So you have to store it to make it available for reuse.

There is an experimental option for `Makefile.PL` called `PERL_STASH_POST_DATA`. If you turn it on, you can get at it again with `$r->subprocess_env("POST_DATA")`. This is not *enabled* by default because it adds a processing overhead for each POST request.

But what do we do with large multipart file uploads? Because `POST` data is not all read in one clump, it's a problem that's not easy to solve in a general way. A transparent way to do this is to switch the request method from POST to GET, and store the POST data in the query string. This handler does exactly this:

```
Apache/POST2GET.pm
------------------
package Apache::POST2GET;
use Apache::Constants qw(M_GET OK DECLINED);

sub handler {
  my $r = shift;
  return DECLINED unless $r->method eq "POST";
  $r->args(scalar $r->content);
  $r->method('GET');
  $r->method_number(M_GET);
  $r->headers_in->unset('Content-length');
  return OK;
}
1;
__END__
```

In *httpd.conf* add:

```
PerlInitHandler Apache::POST2GET
```

or even this:

```
<Limit POST>
  PerlInitHandler Apache::POST2GET
</Limit>
```

To save a few more cycles, so the handler will be called only for POST requests.

Effectively, this trick turns the POST request into a GET request internally. Now when `CGI.pm`, `Apache::Request` or whatever module parses the client data, it can do so more than once since `$r->args` doesn't go away (unless you make it go away by resetting it).

If you are using `Apache::Request`, it solves this problem for you with its instance() class method, which allows `Apache::Request` to be a singleton. This means that whenever you call `Apache::Request`->instance() within a single request you always get the same `Apache::Request` object back.

1.5 Redirecting POST Requests

Under mod_cgi it's not easy to redirect POST requests to some other location. With mod_perl you can easily redirect POST requests. All you have to do is read in the content, set the method to `GET`, populate `args()` with the content to be forwarded and finally do the redirect:

```
use Apache::Constants qw(M_GET);
my $r = shift;
my $content = $r->content;
$r->method("GET");
$r->method_number(M_GET);
$r->headers_in->unset("Content-length");
$r->args($content);
$r->internal_redirect_handler("/new/url");
```

Of course that last line can be any other kind of redirect.

1.6 Redirecting While Maintaining Environment Variables

Let's say you have a module that sets some environment variables.

If you redirect, that's most likely telling the web browser to fetch the new page. This makes it a totally new request, so no environment variables are preserved.

However, if you're using internal_redirect(), you can make the environment variables seen in the sub-process via subprocess_env(). The only nuance is that the `%ENV` keys will be prefixed with `REDIRECT_`.

1.7 Terminating a Child Process on Request Completion

If you want to terminate the child process serving the current request, upon completion of processing anywhere in the code call:

```
$r->child_terminate;
```

Apache won't actually terminate the child until everything it needs to do is done and the connection is closed.

1.8 Setting Content-type and Content-encoding headers in non-OK responses

You cannot set *Content-type* and *Content-encoding* headers in non-OK responses, since Apache overrides these in *http_protocol.c*, *ap_send_error_response()*:

```
r->content_type = "text/html; charset=iso-8859-1";
```

1.9 More on Relative Paths

Many people use relative paths for `require`, `use`, etc., and when they open files in their scripts they make assumptions about the current directory. This will fail if you don't `chdir()` to the correct directory first (as could easily happen if you have another script which calls the first script by its full path).

For example:

```
/home/httpd/perl/test.pl:
-------------------------
#!/usr/bin/perl
open IN, "./foo.txt";
-------------------------
```

This snippet would work if we call the script like this:

```
% chdir /home/httpd/perl
% ./test.pl
```

since `foo.txt` is located in the current directory. But when the current directory isn't */home/httpd/perl*, if we call the script like this:

```
% /home/httpd/perl/test.pl
```

then the script will fail to find `foo.txt`. Think about `crontab`s!

Notice that you cannot use the `FindBin.pm` package, something that you'd do in the regular Perl scripts, because it relies on the `BEGIN` block it won't work under mod_perl. It's loaded and executed only for the first script executed inside the process, all the others will use the cached value, which would be probably incorrect if they reside in different directories.

1.10 Watching the error_log File Without Telneting to the Server

I wrote this script a long time ago, when I had to debug my CGI scripts but didn't have access to the `error_log` file. I asked the admin to install this script and have used it happily since then.

If your scripts are running on these 'Get-free-site' servers, and you cannot debug your script because you can't telnet to the server or can't see the `error_log`, you can ask your sysadmin to install this script.

Note, that it was written for plain Apache, and isn't prepared to handle the complex multiline error and warning messages generated by mod_perl. It also uses a system() call to do the main work with the tail() utility, probably a more efficient perl implementation is due (take a look at `File::Tail` module). You are welcome to fix it and contribute it back to mod_perl community. Thank you!

Here is the code:

```
# !/usr/bin/perl -Tw

use strict;

my $default   = 10;
my $error_log = "/usr/local/apache/logs/error_log";
use CGI;

# untaint $ENV{PATH}
$ENV{'PATH'} = '/bin:/usr/bin';
delete @ENV{'IFS', 'CDPATH', 'ENV', 'BASH_ENV'};

my $q = new CGI;

my $counts = (defined $q->param('count') and $q->param('count'))
  ? $q->param('count') : $default;

print $q->header,
      $q->start_html(-bgcolor => "white",
                     -title   => "Error logs"),
      $q->start_form,
      $q->center(
               $q->b('How many lines to fetch? '),
               $q->textfield('count',10,3,3),
               $q->submit('', 'Fetch'),
               $q->reset,
              ),
      $q->end_form,
      $q->hr;

# untaint $counts
$counts = ($counts =~ /(\d+)/) ? $1 : 0;

print($q->b("$error_log doesn't exist!!!")),exit unless -e $error_log;

open LOG, "tail -$counts $error_log|"
  or die "Can't tail $error_log :$!\n";
my @logs = <LOG>;
close LOG;
  # format and colorize each line nicely

foreach (@logs) {
    s{
     \[(.*?)\]\s* # date
     \[(.*?)\]\s* # type of error
     \[(.*?)\]\s* # client part
     (.*)         # the message
    }
    {
      "[$1] <BR> [".
      colorize($2,$2).
      "] <BR> [$3] <PRE>".
      colorize($2,$4).
      "</PRE>"
    }ex;
```

```
  print "<BR>$_<BR>";
}



#############
sub colorize{
  my ($type,$context) = @_;

  my %colors =
    (
     error  => 'red',
     crit   => 'black',
     notice => 'green',
     warn   => 'brown',
    );

  return exists $colors{$type}
      ? qq{<B><FONT COLOR="$colors{$type}">$context</FONT></B>}
      : $context;
}
```

1.11 Accessing Variables from the Caller's Package

Sometimes you want to access variables from the caller's package. One way is to do something like this:

```
{
  no strict 'vars' ;
  my $caller = caller;
  print qq[$caller --- ${"${caller}::var"}];
}
```

1.12 Handling Cookies

Unless you use some well known module like `CGI::Cookie` or `Apache::Cookie`, you need to handle cookies yourself.

Cookies come in the `$ENV{HTTP_COOKIE}` variable. You can print the raw cookie string as `$ENV{HTTP_COOKIE}`.

Here is a fairly well-known bit of code to take cookie values and put them into a hash:

```
sub get_cookies {
    # cookies are separated by a semicolon and a space, this will
    # split them and return a hash of cookies
  local(@rawCookies) = split (/; /,$ENV{'HTTP_COOKIE'});
  local(%cookies);

  foreach(@rawCookies){
    ($key, $val) = split (/=/,$_);
    $cookies{$key} = $val;
```

```
    }

    return %cookies;
}
```

Or a slimmer version:

```
    sub get_cookies {
        map { split /=/, $_, 2 } split /; /, $ENV{'HTTP_COOKIE'} ;
    }
```

1.13 Sending Multiple Cookies with the Perl API

Given that you have prepared your cookies in `@cookies`, the following code will submit all the cookies:

```
for (@cookies){
 $r->headers_out->add( 'Set-Cookie' => $_ );
}
```

1.14 Sending Cookies in REDIRECT Response

You should use err_headers_out() and not headers_out() when you want to send cookies in the `REDIRECT` response.

```
use Apache::Constants qw(REDIRECT OK);
my $r = shift;
# prepare the cookie in $cookie
$r->err_headers_out->add('Set-Cookie' => $cookie);
$r->headers_out->set(Location => $location);
$r->status(REDIRECT);
$r->send_http_header;
return OK;
```

1.15 Apache::Cookie example: Login Pages by Setting Cookies and Refreshing

On occassion you will need to set a cookie and then redirect the user to another page. This is probably most common when you want a Location to be password protected, and if the user is unauthenticated, display to them a login page, otherwise display another page, but both at the same URL.

1.15.1 Logic

The logic goes something like this:

- Check for login cookie

- If found, display the page
- If not found, display a login page
- Get username/password from a POST
- Authenticate username/password
- If the authentication failed, re-display the login page
- If the authentication passed, set a cookie and redirect to the same page, and display

1.15.2 Example Situation

Let's say that we are writing a handler for the location */dealers* which is a protected area to be accessed only by people who can pass a username / password authentication check.

We will use `Apache::Cookie` here as it runs pretty fast under mod_perl, but `CGI::Cookie` has pretty much the same syntax, so you can use that if you prefer.

For the purposes of this example, we'll assume that we already have any passed parameters in a *%params* hash, the `authenticate()` routine returns **true** or **false**, *display_login()* shows the username and password prompt, and *display_main_page()* displays the protected content.

1.15.2.1 Code

```
if( $params{user} and $params{pass} ) {
    if(!authenticate(%params)) {
```

Authentication failed, send them back to the login page. **NOTE:** It's a good idea to use `no_cache()` to make sure that the client browser doesn't cache the login page.

```
        $r->content_type('text/html');
        $r->no_cache(1);
        $r->send_http_header;
        display_login();
    } else {
```

The user is authenticated, create the cookie with `Apache::Cookie`

```
        my $c = Apache::Cookie->new( $r,
            -name => 'secret',
            -value => 'foo'
            -expires => '+3d',
            -path => '/dealers'
        );
```

NOTE: when setting the 'expires' tag you must set it with *either* a leading `+` or `-`, as if either of these is missing, it will be put literally into the cookie header.

Now send them on their way via the authenticated page

```
        $r->content_type('text/html');
        $c->bake;
        $r->header_out("Refresh"=>"0;url=/dealers");
        $r->no_cache(1);
        $r->send_http_header;
        $r->print( "Authenticated... heading to main page! );
```

The above code will set the headers to refresh (this is the same syntax as for the HTML meta tag) after 0 seconds. The page that is flashed on the screen will have the text in the `$r->print`

```
    }
}
elsif( $cookies{secret} ) {
```

If they already have a secret cookie, display the main (protected) page. Don't forget to check the validity of cookie data!

```
    display_main_page();
}
```

1.16 Passing and Preserving Custom Data Structures Between Handlers

Let's say that you wrote a few handlers to process a request, and they all need to share some custom Perl data structure. The pnotes() method comes to your rescue.

```
# a handler that gets executed first
my %my_data = (foo => 'mod_perl', bar => 'rules');
$r->pnotes('my_data' => \%my_data);
```

The handler prepares the data in hash `%my_data` and calls pnotes() method to store the data internally for other handlers to re-use. All the subsequently called handlers can retrieve the stored data in this way:

```
my $info = $r->pnotes('my_data');
print $info->{foo};
```

prints:

```
mod_perl
```

The stored information will be destroyed at the end of the request.

1.17 Passing Notes Between mod_perl and other (non-Perl) Apache Modules

The notes() method can be used to make various Apache modules talk to each other. In the following snippet the PHP module talks to the mod_perl code (PHP code):

```
if (isset($user) && substr($user,0,1) == "+") {
  apache_note("user", substr($user,1));
  virtual("/internal/getquota");
  $quota      = apache_note("quota");
  $usage_pp   = apache_note("usage_pp");
  $percent_pp = apache_note("percent_pp");
  if ($quota)
    $message .= " | Using $percent_pp% of $quota_pp limit";
}
```

The PHP code sets the *user* and the username pair using the notes mechanism. Then issuing a sub-request to a perl handler:

```
use Apache::Constants qw(REDIRECT OK);
my $r = shift;
my $notes = $r->main->notes();
my ($quota,usage_pp,percent_pp) = getquota($notes->{user}||'');
$r->notes('quota',$quota);
$r->notes('usage_pp',$usage_pp);
$r->notes('percent_pp',$percent_pp);
return OK;
```

which retrieves the username from the notes (using `$r->main->notes`), uses some getquota() function to get the quota related data and then sets the acquired data in the notes for the PHP code. Now the PHP code reads the data from the notes and proceeds with setting `$message` if `$quota` is set.

So any Apache modules can communicate with each other over the Apache notes() mechanism.

You can use notes along with the sub-request methods lookup_uri() and lookup_filename() too. To make it work, you need to set a note in the sub-request. For example if you want to call a php sub-request from within mod_perl and pass it a note, you can do it in the following way:

```
my $subr = $r->lookup_uri('wizard.php3');
$subr->notes('answer' => 42);
$subr->run;
```

As of the time of this writing you cannot access the parent request tables from a PHP handler, therefore you must set this note for the sub-request. Whereas if the sub-request is running in the mod_perl domain, you can always keep the notes in the parent request notes table and access them via the method main():

```
$r->main->notes('answer');
```

1.18 Passing Environment Variables Between Handlers

This is a simple example of passing environment variables between handlers:

Having a configuration:

```
PerlAccessHandler My::Access
PerlLogHandler My::Log
```

and in *startup.pl*:

```
sub My::Access::handler {
  my $r = shift;
  $r->subprocess_env(TICKET => $$);
  $r->notes(TICKET => $$);
}

sub My::Log::handler {
  my $r = shift;
  my $env = $r->subprocess_env('TICKET');
  my $note = $r->notes('TICKET');
  warn "env=$env, note=$note\n";
}
```

Adding `%{TICKET}e` and `%{TICKET}n` to the `LogFormat` for access_log works fine too.

1.19 Verifying Whether A Request Was Received Over An SSL Connection

Just like `$ENV{MODPERL}` is checked to see whether the code is run under mod_perl, `$ENV{HTTPS}` is set by ssl modules and therefore can be used to check whether a request is running over SSL connection. For example:

```
print "SSL" if $ENV{HTTPS};
```

If `PerlSetupEnv Off` setting is in effect, `$ENV{HTTPS}` won't be available, and then:

```
print "SSL" if $r->subprocess_env('https');
```

should be used instead.

Note that it's also possible to check the scheme:

```
print "SSL" if Apache::URI->parse($r)->scheme =~ m/^https/;
```

but it's not one hundred percent certain unless you control the server and you know that you run a secure server on the port 443.

1.20 CGI::params in the mod_perl-ish Way

You can retrieve the request parameters in a similar to `CGI::params` way using this technique:

```
my $r = shift;  # or $r = Apache->request
my %params = $r->method eq 'POST' ? $r->content : $r->args;
```

assuming that all your variables are single key-value pairs.

Also take a look at `Apache::Request` which has the same API as `CGI.pm` for extracting and setting request parameters.

1.21 Subclassing Apache::Request

To subclass a package you simply modify @ISA, for example:

```
package My::TestAPR;

use strict;
use vars qw/@ISA/;
@ISA = qw/Apache::Request/;

sub new {
      my ($proto, $apr) = @_;
      my $class = ref($proto) || $proto;
      bless { _r => $apr }, $class;
}

sub param {
      my ($self, $key) = @_;
      my $apr = $self->{_r};
      # Here we are calling the Apache::Request object's param method
      $apr->param($key);
}

sub sum {
      my ($self, $key) = @_;
      my $apr = $self->{_r};
      my @values = $apr->param($key);
      my $sum = 0;
      for (@values) {
              $sum += $_;
      }
      $sum;
}
1;
__END__
```

1.22 Sending Email from mod_perl

There is nothing special about sending email from mod_perl, it's just that we do it a lot. There are a few important issues. The most widely used approach is starting a `sendmail` process and piping the headers and the body to it. The problem is that `sendmail` is a very heavy process and it makes mod_perl processes less efficient.

If you don’t want your process to wait until delivery is complete, you can tell `sendmail` not to deliver the email straight away, but either do it in the background or just queue the job until the next queue run. This can significantly reduce the delay for the mod_perl process which would otherwise have to wait for the `sendmail` process to complete. This can be specified for all deliveries in *sendmail.cf* or on each invocation on the sendmail command line:

- `-odb` (deliver in the background)
- `-odq` (queue-only) or
- `-odd` (queue, and also defer the DNS/NIS lookups).

The trend is to move away from sendmail(1) and switch to using lighter mail delivery programs like qmail(1) or postfix(1). You should check the manpage of your favorite mailer application for equivalent configuration presented for sendmail(1).

The most efficient approach is to talk directly to the SMTP server. Luckily `Net::SMTP` modules makes this very easy. The only problem is when `Net::SMTP` fails to deliver the mail, because the destination peer server is temporarily down. But from the other side `Net::SMTP` allows you to send email much faster, since you don’t have to invoke a dedicated process. Here is an example of a subroutine that sends email.

```
use Net::SMTP ();
use Carp qw(carp verbose);

#
# Sends email by using the SMTP Server
#
# The SMTP server as defined in Net::Config
# Alternatively you can hardcode it here, look for $smtp_server below
#
sub send_mail{
  my ($from, $to, $subject, $body) = @_;

  carp "From missing" unless defined $from ; # Prefer to exit early if errors
  carp "To missing"   unless defined $to ;

  my $mail_message = <<__END_OF_MAIL__;
To: $to
From: $from
Subject: $subject

$body

__END_OF_MAIL__

    # Set this parameter if you don’t have a valid Net/Config.pm
    # entry for SMTP host and uncomment it in the Net::SMTP->new
    # call
  # my $smtp_server = ’localhost’;

    # init the server
  my $smtp = Net::SMTP->new(
```

```
              # $smtp_server,
              Timeout => 60,
              Debug   => 0,
             );

  $smtp->mail($from) or carp ("Failed to specify a sender [$from]\n");
  $smtp->to($to) or carp ("Failed to specify a recipient [$to]\n");
  $smtp->data([$mail_message]) or carp ("Failed to send a message\n");

  $smtp->quit or carp ("Failed to quit\n");

} #  end of sub send_mail
```

1.23 A Simple Handler To Print The Environment Variables

The code:

```
package MyEnv;
use Apache;
use Apache::Constants;
sub handler{
  my $r = shift;
  print $r->send_http_header("text/plain");
  print map {"$_ => $ENV{$_}\n"} keys %ENV;
  return OK;
}
1;
```

The configuration:

```
PerlModule MyEnv
<Location /env>
  SetHandler perl-script
  PerlHandler MyEnv
</Location>
```

The invocation:

```
http://localhost/env
```

1.24 mod_rewrite in Perl

We can easily implement everything mod_rewrite does in Perl. We do this with help of PerlTransHandler, which is invoked at the beginning of request processing. For example consider that we need to perform a redirect based on query string and URI, the following handler does that.

```
package Apache::MyRedirect;
use Apache::Constants qw(OK REDIRECT);
use constant DEFAULT_URI => 'http://www.example.org';

sub handler {
  my $r    = shift;
  my %args = $r->args;
```

```
    my $path = $r->uri;

    my $uri = (($args{'uri'}) ? $args{'uri'} : DEFAULT_URI) . $path;

    $r->header_out(Location => $uri);
    $r->status(REDIRECT);
    $r->send_http_header;

    return OK;
  }
```

Set it up in *httpd.conf* as:

```
PerlTransHandler Apache::MyRedirect
```

The code consists of three parts: request data retrieval, deciding what to do based on this data and finally setting the headers and the status and issuing redirect.

So if a client submits a request of this kind:

```
http://www.example.com/news/?uri=http://www2.example.com/
```

`$uri` will hold *http://www2.example.com/news/* and that's where the request will be redirected.

1.25 URI Rewrite in PerlTransHandler

Suppose that before a content handler is invoked you want to make this translation:

```
/articles/10/index.html  =>  /articles/index.html?id=10
```

This *TransHandler* will do that for you:

```
My/Trans.pm
-----------
package My::Trans;
use Apache::Constants qw(:common);
sub handler {
  my $r = shift;
  my $uri = $r->uri;
  my ($id) = ($uri =~ m|^/articles/(.*?)/|);
  $r->uri("/articles/index.html");
  $r->args("id=$id");
  return DECLINED;
}
1;
```

and in *httpd.conf*:

```
PerlModule My::Trans
PerlTransHandler My::Trans
```

The handler code retrieves the request object and the URI. Then it retrieves the *id* using the regular expression. Finally it sets the new value of the URI and the arguments string. The handler returns `DECLINED` so the default Apache transhandler will take care of URI to filename remapping.

Notice the technique to set the arguments. By the time the Apache-request object has been created, arguments are handled in a separate slot, so you cannot just push them into the original URI. Therefore the args() method should be used.

1.26 Setting PerlHandler Based on MIME Type

It's very easy to implement a dispatching module based on the MIME type of request. So a different content handler will be called for a different MIME type. This is an example of such a dispatcher:

```
package My::MimeTypeDispatch;
use Apache::Constants qw(DECLINED);

my %mime_types = (
  'text/html'  => \&HTML::Template::handler,
  'text/plain' => \&My::Text::handler,
);

sub handler {
  my $r = shift;
  if (my $h = $mime_types{$r->content_type}) {
    $r->push_handlers(PerlHandler => $h);
    $r->handler('perl-script');
  }
return DECLINED;
}
1;
__END__
```

And in *httpd.conf* we add:

```
PerlFixupHandler My::MimeTypeDispatch
```

After declaring the package name and importing constants, we set a translation table of MIME types and corresponding handlers to be called. Then comes the handler, where the request object is retrieved and if its MIME type is found in our translation table we set the handler that should handle this request. Otherwise we do nothing. At the end we return `DECLINED` so some other fixup handler could take over.

1.27 SSI and Embperl -- Doing Both

This handler lets you use both SSI and Embperl in the same request:

Use it in a `<FilesMatch>` Section or similar:

```
PerlModule Apache::EmbperlFilter Apache::SSI
<FilesMatch "\.epl">
   PerlSetVar Filter On
   PerlHandler Apache::EmbperlFilter Apache::SSI
</FilesMatch>

package Apache::EmbperlFilter;

use Apache::Util qw(parsedate);
use HTML::Embperl;
use Apache::SSI ();
use Apache::Constants;

use strict;
use vars qw($VERSION);

$VERSION = '0.03';
my ($r, %param, $input, $output);

sub handler {
    $r = shift;
    my ($fh, $status) = $r->filter_input();
    unless ($status == OK) {
        return $status
    }
    local $/ = undef;
    $input = scalar(<$fh>);
    %param = ();
    $param{input} = \$input;
    $param{req_rec} = $r;
    $param{output} = \$output;
    $param{mtime} = mtime();
    $param{inputfile} = $r->filename();
    HTML::Embperl::ScanEnvironement(\%param);
    HTML::Embperl::Execute(\%param);
    print $output;
    return OK;
}

sub mtime {
    my $mtime = undef;
    if (my $last_modified = $r->headers_out->{'Last-Modified'}) {
        $mtime = parsedate $last_modified;
    }
    $mtime;
}


1;
__END__
```

1.28 Getting the Front-end Server's Name in the Back-end Server

Assume that you have more than one front-end server, and you want to dynamically figure out the front-end server name in the back-end server. mod_proxy and mod_rewrite provide the solution.

Compile apache with both mod_proxy and mod_rewrite, then use a directive something like this:

```
RewriteEngine On
RewriteLog /somewhere/rewrite.log
RewriteLogLevel 3
RewriteRule ^/foo/bar(.*)$ \
http://example.com:8080/foo/bar/$1?IP=%{REMOTE_HOST} [QSA,P]
```

This will have all the urls starting with */some/url* proxied off to the other server at the same url. It will append the `REMOTE_HOST` header as a query string argument. (QSA = Query String Append, P = Proxy). There is probably a way to remap it as an X-Header of some sort, but if query string is good enough for you, then this should work really nicely.

1.29 Authentication Snippets

Getting the authenticated username: `$r->connection->user()`, or `$ENV{REMOTE_USER}` if you're in a CGI emulation.

Example:

```
my $r = shift;

my ($res, $sent_pwd) = $r->get_basic_auth_pw;
return $res if $res; #decline if not Basic

my $user = $r->connection->user;
```

1.30 Emulating the Authentication Mechanism

You can provide your own mechanism to authenticate users, instead of the standard one. If you want to make Apache think that the user was authenticated by the standard mechanism, set the username with:

```
$r->connection->user('username');
```

Now you can use this information for example during the logging, so that you can have your "username" passed as if it was transmitted to Apache through HTTP authentication.

1.31 An example of using Apache::Session::DBI with cookies

META: should be annotated at some point. (an example was posted to the mod_perl list)

```
use strict;
use DBI;
use Apache::Session::DBI;
use CGI;

# [...]

# Initiate a session ID
my $session = ();
my $opts = {  autocommit => 0,
              lifetime   => 3600 };     # 3600 is one hour

# Read in the cookie if this is an old session
my $r = Apache->request;
my $no_cookie = '';
my $cookie = $r->header_in('Cookie');
{
    # eliminate logging from Apache::Session::DBI's use of 'warn'
    local $^W = 0;

    if (defined($cookie) && $cookie ne '') {
        $cookie =~ s/SESSION_ID=(\w*)/$1/;
        $session = Apache::Session::DBI->open($cookie, $opts);
        $no_cookie = 'Y' unless defined($session);
    }
    # Could have been obsolete - get a new one
    $session = Apache::Session::DBI->new($opts) unless defined($session);
}

# Might be a new session, so let's give them a cookie back
if (! defined($cookie) || $no_cookie) {
    local $^W = 0;

    my $session_cookie = "SESSION_ID=$session->{'_ID'}";
    $r->header_out("Set-Cookie" => $session_cookie);
}
```

1.32 Using DESTROY to Finalize Output

Well, as always with Perl -- TMTOWTDI (There's More Than One Way To Do It), one of the readers is using DESTROY to finalize output, and as a cheap means of buffering.

```
package buffer;
use Apache;

sub new {
   my $class = shift;
   my $self = bless {
       'r' => shift,
       'message' => ""
```

```
    }, $class;
    $self->{apr} = Apache::Request->new($self->{r},
                                        POST_MAX=>(32*1024));
    $self->content_type('text/plain');
    $self->{r}->no_cache(1);
}

sub message {
      my $self = shift;
      $self->{message} .= join("\n", @_);
}

sub DESTROY {
      my $self = shift;
      $self->{apr}->send_http_header;
      $self->{apr}->print($self->{message});
}
1;
```

Now you can have perl scripts like:

```
use buffer;
my $b = new buffer(shift);

$b->message(p("Hello World"));
# end
```

and save a bunch of duplicate code across otherwise inconvenient gaggles of small scripts.

But suppose you also want to redirect the client under some circumstances, and send the HTTP status code 302. You might try this:

```
sub redir {
      my $self = shift;
      $self->{redirect} = shift;
      exit;
}
```

and re-code DESTROY as:

```
sub DESTROY {
      my $self = shift;
      if ($self->{redirect}) {
              $self->{apr}->status{REDIRECT};
              $self->{apr}->header_out("Location", $self->{redirect});
              $self->{apr}->send_http_header;
              $self->{apr}->print($self->{redirect});
      } else {
              $self->{apr}->send_http_header;
              $self->{apr}->print($self->{message});
      }
}
```

But you'll find that while the browser redirects itself, mod_perl logs the result code as 200. It turns out that status() only touches the Apache response, and the log message is determined by the Apache return code.

Aha! So we'll change the exit() in redir() to exit(REDIRECT). This fixes the log code, but causes a bogus *"[error] 302"* line in the *error_log*. That comes from `Apache::Registry`:

```
my $errsv = "";
if($@) {
    $errsv = $@;
    $@ = ''; #XXX fix me, if we don't do this Apache::exit() breaks
    $@{$uri} = $errsv;
}

if($errsv) {
    $r->log_error($errsv);
    return SERVER_ERROR unless $Debug && $Debug & 2;
    return Apache::Debug::dump($r, SERVER_ERROR);
}
```

So you see that any time the return code causes `$@` to return true, we'll get an error line. Not wanting this, what can we do?

We can hope that a future version of mod_perl will allow us to set the HTTP result code independent from the handler return code (perhaps a log_status() method? or at least an `Apache::LOG_HANDLER_RESULT` config variable?).

In the meantime, there's `Apache::RedirectLogFix`, distributed with mod_perl.

Add to your *httpd.conf*:

```
PerlLogHandler Apache::RedirectLogFix
```

and take a look at the source code below. Note that it requires us to return the HTTP status code 200.

```
package Apache::RedirectLogFix;

use Apache::Constants qw(OK DECLINED REDIRECT);

sub handler {
   my $r = shift;
   return DECLINED unless $r->handler && ($r->handler eq "perl-script");

   if(my $loc = $r->header_out("Location")) {
       if($r->status == 200 and substr($loc, 0, 1) ne "/") {
           $r->status(REDIRECT);
           return OK
       }
   }
   return DECLINED;
}

1;
```

Now, if we wanted to do the same sort of thing for an error 500 handler, we could write another `PerlLogHandler` (call it `ServerErrorLogFix`). But we'll leave that as an exercise for the reader, and hope that it won't be needed in the next mod_perl release. After all, it's a little awkward to need a `LogHandler` to clean up after ourselves....

1.33 Passing Arguments to a SSI script

Consider the following `Apache::Include` snippet:

```
<!--#perl sub="Apache::Include" arg="/perl/ssi.pl" -->
```

Now if you want to pass arguments, you cannot do that with `Apache::Include`. The solution is to define a subroutine that's pulled in at the startup:

```
sub My::ssi {
    my($r, $one, $two, $three) = @_;
   ...
}
```

In the html file:

```
<!--#perl sub="My::ssi" arg="one" arg="two" arg="three" -->
```

1.34 Setting Environment Variables For Scripts Called From CGI.

Perl uses `sh()` for its `system()` and `open()` calls. So if you want to set a temporary variable when you call a script from your CGI you do something like this:

```
open UTIL, "USER=stas ; script.pl | " or die "...: $!\n";
```

or

```
system "USER=stas ; script.pl";
```

This is useful, for example, if you need to invoke a script that uses CGI.pm from within a mod_perl script. We are tricking the Perl script into thinking it's a simple CGI, which is not running under mod_perl.

```
open(PUBLISH, "GATEWAY_INTERFACE=CGI/1.1 ; script.cgi
     \"param1=value1&param2=value2\" |") or die "...: $!\n";
```

Make sure that the parameters you pass are shell safe -- all "unsafe" characters like single-quote and back-tick should be properly escaped.

Unfortunately mod_perl uses fork() to run the script, so you have probably thrown out the window most of the performance gained from using mod_perl. To avoid the fork, change script.cgi to a module containing a subroutine which you can then call directly from your mod_perl script.

1.35 Mysql Backup and Restore Scripts

This is somewhat off-topic, but since many of us use mysql or some other RDBMS in their work with mod_perl driven sites, it's good to know how to backup and restore the databases in case of database corruption.

First we should tell mysql to log all the clauses that modify the databases (we don't care about SELECT queries for database backups). Modify the `safe_mysql` script by adding the *--log-update* options to the `mysql` server startup parameters and restart the server. From now on all the non-select queries will be logged to the */var/lib/mysql/www.bar.com* logfile. Your hostname will show up instead of *www.bar.com*.

Now create a *dump* directory under */var/lib/mysql/*. That's where the backups will be stored (you can name the directory as you wish of course).

Prepare the backup script and store it in a file, e.g: */usr/local/sbin/mysql/mysql.backup.pl*

This is the original code *code/mysql-3.22.29_backup.pl*:

```
#!/usr/bin/perl -w

# this script should be run from the crontab every night or in shorter
# intervals. This scripts does a few things.
# 1. dump all the tables into a separate dump files (these dump files
# are ready for DB restore)
# 2. backups the last update log file and create a new log file


use strict;
my $data_dir = "/var/lib/mysql";
my $update_log = "$data_dir/www.bar.com";
my $dump_dir  = "$data_dir/dump";
my $gzip_exec = "/bin/gzip";
my @db_names = qw(bugs mysql bonsai);
my $mysql_admin_exec = "/usr/bin/mysqladmin ";

    # convert unix time to date + time
my ($sec,$min,$hour,$mday,$mon,$year) = localtime(time);
my $time  = sprintf("%0.2d:%0.2d:%0.2d",$hour,$min,$sec);
my $date  = sprintf("%0.2d.%0.2d.%0.4d",++$mon,$mday,$year+1900);
my $timestamp = "$date.$time";

# dump all the DBs we want to backup
foreach my $db_name (@db_names) {
  my $dump_file = "$dump_dir/$timestamp.$db_name.dump";
  my $dump_command = "/usr/bin/mysqldump -c -e -l -q --flush-logs $db_name > $dump_file";
  system $dump_command;
}

# move update log to backup for later restore if needed
rename $update_log, "$dump_dir/$timestamp.log" if -e $update_log;

# restart the update log to log to a new file!
```

```
‘/usr/bin/mysqladmin refresh‘;

# compress all the created files
system "$gzip_exec $dump_dir/$timestamp.*";
```

This is the code modified to work with mysql-3.22.30+ *code/mysql-3.22.30+_backup.pl*:

```
#!/usr/bin/perl -w

# this script should be run from the crontab every night or in shorter
# intervals. This scripts does a few things.
# 1. dump all the tables into a separate dump files (these dump files
# are ready for DB restore)
# 2. backups the last update log file and create a new log file

#This script originates from the perl.apache.org site, but I have adapted it to work
#properly with the newer versions of MySQL, where the log files are named differently
#WVW 14/02/2000 w@ba.be

use strict;

my $data_dir = "/var/lib/mysql";
my $update_log = "$data_dir/central2.001";
my $dump_dir  = "$data_dir/backup";
my $gzip_exec = "/bin/gzip";
my @db_names = qw(mysql besup);
my $mysql_admin_exec = "/usr/bin/mysqladmin ";
my $hostname = "central2";

my $password = "babedb";

# convert unix time to date + time
my ($sec,$min,$hour,$mday,$mon,$year) = localtime(time);
my $time  = sprintf("%0.2d:%0.2d:%0.2d",$hour,$min,$sec);
my $date  = sprintf("%0.2d.%0.2d.%0.4d",++$mon,$mday,$year+1900);
my $timestamp = "$date.$time";

# dump all the DBs we want to backup
foreach my $db_name (@db_names) {
  my $dump_file = "$dump_dir/$timestamp.$db_name.dump";
  my $dump_command = "/usr/bin/mysqldump -c -e -l -q --flush-logs -p$password $db_name > $dump_file";
  system $dump_command;
}

mkdir "$dump_dir/$timestamp.log", 0;
‘mv $data_dir/$hostname.[0-9]* $dump_dir/$timestamp.log‘;

# move update log to backup for later restore if needed
#rename $update_log, "$dump_dir/$timestamp.log" if -e $update_log;

# restart the update log to log to a new file!
‘/usr/bin/mysqladmin refresh -p$password‘;

# compress all the created files
system "$gzip_exec $dump_dir/$timestamp.log/*";
system "$gzip_exec $dump_dir/$timestamp.*.dump*";
```

You might need to change the executable paths according to your system. List the names of the databases you want to backup using the `db_names` array.

Here is another version using `File::Backup`:

```
#!/usr/bin/perl
# written by Miroslav Madzarevic, mire@modperldev.com
use strict;

umask 0177;

use File::Backup qw|backup|;

backup(
   'from'          => "",
   'to'            => "/opt/backup/mysql/backup",
   'torootname'    => "example_backup_",
   'keep'          => 4,
   'tar'           => "/usr/bin/mysqldump",
   'compress'      => "/usr/bin/bzip2",
   'tarflags'      => "example -uroot -proot_pass -a >",
   'compressflags' => "",
   'tarsuffix'     => '.sql',
);
```

Now make the script executable and arrange the crontab entry to run the backup script nightly. Note that the disk space used by the backups will grow without bound and you should remove the old backups. Here is a sample crontab entry to run the script at 4am every day:

```
0 4 * * * /usr/local/sbin/mysql/mysql.backup.pl > /dev/null 2>&1
```

So now at any moment we have the dump of the databases from the last execution of the backup script and the log file of all the clauses that have updated the databases since then. If the database gets corrupted we have all the information to restore it to the state it was in at our last backup. We restore it with the following script, which I put in: */usr/local/sbin/mysql/mysql.restore.pl*

This is the original code *code/mysql-3.22.29_restore.pl*:

```
#!/usr/bin/perl -w

# this scripts restores the DBs

# Usage: mysql.restore.pl update.log.gz dump.db1.gz [... dump.dbn.gz]
# all files dump* are compressed as we expect them to be created by
# mysql.backup utility

# example:
# % mysql.restore.pl myhostname.log.gz 12.10.1998.16:37:12.*.dump.gz

# .dump.gz extension.

use strict;

use FindBin qw($Bin);

my $data_dir   = "/var/lib/mysql";
my $dump_dir   = "$data_dir/dump";
my $gzip_exec  = "/bin/gzip";
```

```
my $mysql_exec = "/usr/bin/mysql -f ";
my $mysql_backup_exec = "$Bin/mysql.backup.pl";
my $mysql_admin_exec  = "/usr/bin/mysqladmin ";

my $update_log_file = '';
my @dump_files = ();

# split input files into an update log and the dump files
foreach (@ARGV) {
  push(@dump_files, $_),next unless /\.log\.gz/;
  $update_log_file = $_;
}

die "Usage: mysql.restore.pl update.log.gz dump.db1.gz [... dump.dbn.gz]\n"
  unless defined @dump_files and @dump_files > 0;

# load the dump files
foreach (@dump_files) {

    # check the file exists
  warn("Can't locate $_"),next unless -e $_;

    # extract the db name from the dump file
  my $db_name = $1 if /\d\d\.\d\d.\d\d.\d\d:\d\d:\d\d\.(\w+)\.dump\.gz/;

  warn("Can't extract DB name from the file name,
        probably an error in the file format"),
          next unless defined $db_name and $db_name;

    # we want to drop the table since restore will rebuild it!
    # force to drop the db without confirmation
  my $drop_command = "$mysql_admin_exec -f drop $db_name";
  system $drop_command;

  $drop_command = "$mysql_admin_exec create $db_name";
  system $drop_command;

    # build the command and execute it
  my $restore_command = "$gzip_exec -cd $_ | $mysql_exec $db_name";
  system $restore_command;
}

# now load the update_log file (update the db with the changes since
# the last dump
warn("Can't locate $update_log_file"),next unless  -e $update_log_file;

my $restore_command =
  "$gzip_exec -cd $update_log_file |$mysql_exec";
system $restore_command;

# rerun the mysql.backup.pl since we have reloaded the dump files
# and update log , and we must rebuild backups!
system $mysql_backup_exec;
```

This is the code modified to work with mysql-3.22.30+ *code/mysql-3.22.30+_restore.pl*:

```
#!/usr/bin/perl -w

# this scripts restores the DBs

# Usage: mysql.restore.pl update.log.gz dump.db1.gz [... dump.dbn.gz]
# all files dump* are compressed as we expect them to be created by
# mysql.backup utility

# example:
# % mysql.restore.pl myhostname.log.gz 12.10.1998.16:37:12.*.dump.gz

# .dump.gz extension.

use strict;

use FindBin qw($Bin);

my $data_dir   = "/var/lib/mysql";
my $dump_dir   = "$data_dir/backup";
my $gzip_exec  = "/bin/gzip";
my $mysql_exec = "/usr/bin/mysql -f -pbabedb";
my $mysql_backup_exec = "$Bin/mysql_backup.pl";
my $mysql_admin_exec  = "/usr/bin/mysqladmin -pbabedb";

my $update_log_dir = '';
my @dump_files = ();

# split input files into an update log and the dump files
foreach (@ARGV) {
  push(@dump_files, $_),next unless /\.log/;
  $update_log_dir = $_;
}

die "Usage: mysql.restore.pl update.log.dir dump.db1.gz [... dump.dbn.gz]\n"
  unless defined @dump_files and @dump_files > 0;

# load the dump files
foreach (@dump_files) {

  # check the file exists
  warn("Can't locate $_"),next unless -e $_;

  # extract the db name from the dump file
  my $db_name = $1 if /\d\d\.\d\d.\d\d\d\d.\d\d:\d\d:\d\d\.(\w+)\.dump\.gz/;

  warn("Can't extract DB name from the file name,
        probably an error in the file format"),
        next unless defined $db_name and $db_name;

  # we want to drop the table since restore will rebuild it!
  # force to drop the db without confirmation
  my $drop_command = "$mysql_admin_exec -f drop $db_name";
  system $drop_command;

  $drop_command = "$mysql_admin_exec create $db_name";
```

```
  system $drop_command;

  # build the command and execute it
  my $restore_command = "$gzip_exec -cd $_ | $mysql_exec $db_name";
  system $restore_command;
}

# now load the update_log file (update the db with the changes since
# the last dump
warn("Can't locate $update_log_dir"),next unless  -d $update_log_dir;

my $restore_command =
  "$gzip_exec -cd $update_log_dir/* |$mysql_exec";
system $restore_command;

# rerun the mysql.backup.pl since we have reloaded the dump files
# and update log , and we must rebuild backups!
system $mysql_backup_exec;
```

These are kinda dirty scripts, but they work... if you come up with cleaner scripts, please contribute them... thanks

Update: there is now a "mysqlhotcopy" utility distributed with MySQL that can make an atomic snapshot of a database. (by Tim Bunce) So you may consider using it instead.

1.36 Maintainers

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

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

1.37 Authors

- Stas Bekman <stas (at) stason.org>
- Alan Bailward, <alan (at) ufies.org>

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

Table of Contents: