1 HTTP Handlers

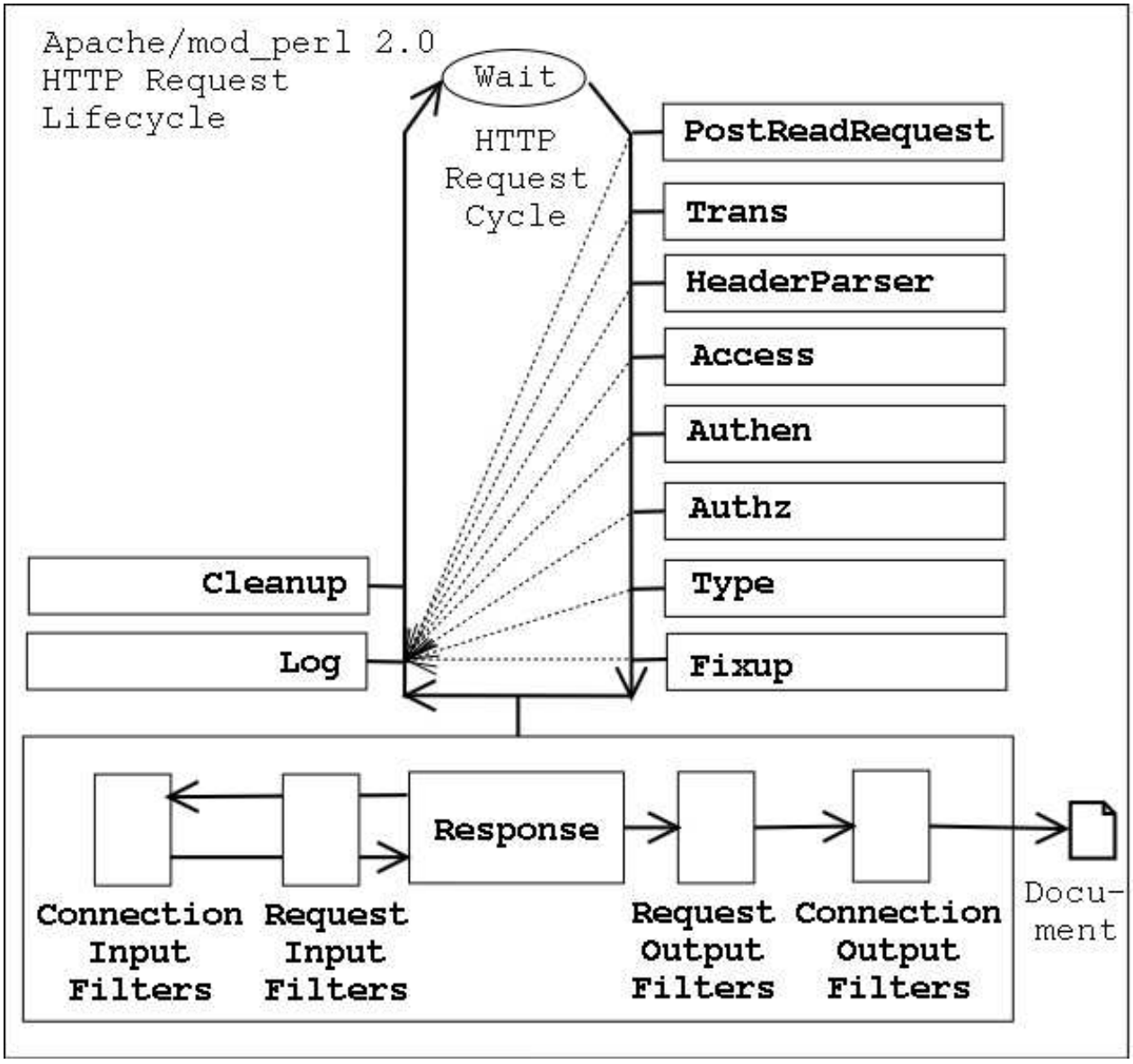
1.1 Description

This chapter explains how to implement the HTTP protocol handlers in mod_perl.

1.2 HTTP Request Cycle Phases

Those familiar with mod_perl 1.0 will find the HTTP request cycle in mod_perl 2.0 to be almost identical to the mod_perl 1.0's model. The only difference is in the *response* phase which now includes filtering. Also the `PerlHandler` directive has been renamed to `PerlResponseHandler` to better match the corresponding Apache phase name (*response*).

The following diagram depicts the HTTP request life cycle and highlights which handlers are available to mod_perl 2.0:



From the diagram it can be seen that an HTTP request is processes by 11 phases, executed in the following order:

1. **PerlPostReadRequestHandler (PerlInitHandler)**
2. **PerlTransHandler**
3. **PerlMapToStorageHandler**
4. **PerlHeaderParserHandler (PerlInitHandler)**
5. **PerlAccessHandler**
6. **PerlAuthenHandler**
7. **PerlAuthzHandler**
8. **PerlTypeHandler**
9. **PerlFixupHandler**

10. **PerlResponseHandler**
11. **PerlLogHandler**
12. **PerlCleanupHandler**

It's possible that the cycle will not be completed if any of the phases terminates it, usually when an error happens. In that case Apache skips to the logging phase (mod_perl executes all registered `PerlLogHandler` handlers) and finally the cleanup phase happens.

Notice that when the response handler is reading the input data it can be filtered through request input filters, which are preceded by connection input filters if any. Similarly the generated response is first run through request output filters and eventually through connection output filters before it's sent to the client. We will talk about filters in detail later in this chapter.

Before discussing each handler in detail remember that if you use stacked handlers feature (META: add link to where it's discussed [go read 1.0 docs for now, as it works the same]) all handlers in the chain will be run as long as they return `Apache::OK` or `Apache::DECLINED`. Because stacked handlers is a special case. So don't be surprised if you've returned `Apache::OK` and the next handler was still executed. This is a feature, not a bug.

Now let's discuss each of the mentioned handlers in detail.

1.2.1 PerlPostReadRequestHandler

The *post_read_request* phase is the first request phase and happens immediately after the request has been read and HTTP headers were parsed.

This phase is usually used to do processing that must happen once per request. For example `Apache::Reload` is usually invoked at this phase to reload modified Perl modules.

This phase is of type `RUN_ALL`.

The handler's configuration scope is `SRV`, because at this phase the request has not yet been associated with a particular filename or directory.

Now, let's look at an example. Consider the following registry script:

```
touch.pl
--------
use strict;
use warnings;

use Apache::ServerUtil ();
use File::Spec::Functions qw(catfile);

my $r = shift;
$r->content_type('text/plain');
```

```
my $conf_file = catfile Apache::server_root_relative($r->pool, 'conf'),
    "httpd.conf";

printf "$conf_file is %0.2f minutes old", 60*24*(-M $conf_file);
```

This registry script is supposed to print when the last time *httpd.conf* has been modified, compared to the start of the request process time. If you run this script several times you might be surprised that it reports the same value all the time. Unless the request happens to be served by a recently started child process which will then report a different value. But most of the time the value won't be reported correctly.

This happens because the `-M` operator reports the difference between file's modification time and the value of a special Perl variable `$^T`. When we run scripts from the command line, this variable is always set to the time when the script gets invoked. Under mod_perl this variable is getting preset once when the child process starts and doesn't change since then, so all requests see the same time, when operators like `-M`, `-C` and `-A` are used.

Armed with this knowledge, in order to make our code behave similarly to the command line programs we need to reset `$^T` to the request's start time, before `-M` is used. We can change the script itself, but what if we need to do the same change for several other scripts and handlers? A simple `PerlPostRead-RequestHandler` handler, which will be executed as the very first thing of each requests, comes handy here:

```
file:MyApache/TimeReset.pm
--------------------------
package MyApache::TimeReset;

use strict;
use warnings;

use Apache::RequestRec ();

use Apache::Const -compile => 'OK';

sub handler {
    my $r = shift;
    $^T = $r->request_time;
    return Apache::OK;
}
1;
```

We could do:

```
$^T = time();
```

But to make things more efficient we use `$r->request_time` since the request object `$r` already stores the request's start time, so we get it without performing an additional system call.

To enable it just add to *httpd.conf*:

```
PerlPostReadRequestHandler MyApache::TimeReset
```

either to the global section, or to the <VirtualHost> section if you want this handler to be run only for a specific virtual host.

1.2.2 PerlTransHandler

The *translate* phase is used to perform the translation of a request's URI into an corresponding filename. If no custom handler is provided, the server's standard translation rules (e.g., Alias directives, mod_rewrite, etc.) will continue to be used. A PerlTransHandler handler can alter the default translation mechanism or completely override it.

In addition to doing the translation, this stage can be used to modify the URI itself and the request method. This is also a good place to register new handlers for the following phases based on the URI.

This phase is of type RUN_FIRST.

The handler's configuration scope is SRV, because at this phase the request has not yet been associated with a particular filename or directory.

There are many useful things that can be performed at this stage. Let's look at the example handler that rewrites request URIs, similar to what mod_rewrite does. For example, if your web-site was originally made of static pages, and now you have moved to a dynamic page generation chances are that you don't want to change the old URIs, because you don't want to break links for those who link to your site. If the URI:

```
http://example.com/news/20021031/09/index.html
```

is now handled by:

```
http://example.com/perl/news.pl?date=20021031&id=09&page=index.html
```

the following handler can do the rewriting work transparent to *news.pl*, so you can still use the former URI mapping:

```
file:MyApache/RewriteURI.pm
---------------------------
package MyApache::RewriteURI;

use strict;
use warnings;

use Apache::RequestRec ();

use Apache::Const -compile => qw(DECLINED);

sub handler {
    my $r = shift;

    my ($date, $id, $page) = $r->uri =~ m|^/news/(\d+)/(\d+)/(.*)|;
    $r->uri("/perl/news.pl");
    $r->args("date=$date&id=$id&page=$page");
```

```
    return Apache::DECLINED;
}
1;
```

The handler matches the URI and assigns a new URI via `$r->uri()` and the query string via `$r->args()`. It then returns `Apache::DECLINED`, so the next translation handler will get invoked, if more rewrites and translations are needed.

Of course if you need to do a more complicated rewriting, this handler can be easily adjusted to do so.

To configure this module simply add to *httpd.conf*:

```
PerlTransHandler +MyApache::RewriteURI
```

1.2.3 PerlMapToStorageHandler META: add something here

This phase is of type `RUN_FIRST`.

The handler's configuration scope is `SRV`, because at this phase the request has not yet been associated with a particular filename or directory.

1.2.4 PerlHeaderParserHandler

The *header_parser* phase is the first phase to happen after the request has been mapped to its `<Location>` (or an equivalent container). At this phase the handler can examine the request headers and to take a special action based on these. For example this phase can be used to block evil clients targeting certain resources, while little resources were wasted so far.

This phase is of type `RUN_ALL`.

The handler's configuration scope is `DIR`.

This phase is very similar to `PerlPostReadRequestHandler`, with the only difference that it's run after the request has been mapped to the resource. Both phases are useful for doing something once per request, as early as possible. And usually you can take any `PerlPostReadRequestHandler` and turn it into `PerlHeaderParserHandler` by simply changing the directive name in *httpd.conf* and moving it inside the container where it should be executed. Moreover, because of this similarity mod_perl provides a special directive `PerlInitHandler` which if found outside resource containers behaves as `PerlPostReadRequestHandler`, otherwise as `PerlHeaderParserHandler`.

You already know that Apache handles the `HEAD`, `GET`, `POST` and several other HTTP methods. But did you know that you can invent your own HTTP method as long as there is a client that supports it. If you think of emails, they are very similar to HTTP messages: they have a set of headers and a body, sometimes a multi-part body. Therefore we can develop a handler that extends HTTP by adding a support for the `EMAIL` method. We can enable this protocol extension and push the real content handler during the `PerlHeaderParserHandler` phase:

```
<Location /email>
    PerlHeaderParserHandler MyApache::SendEmail
</Location>
```

and here is the `MyApache::SendEmail` handler:

```
file:MyApache/SendEmail.pm
--------------------------
package MyApache::SendEmail;

use strict;
use warnings;

use Apache::RequestRec ();
use Apache::RequestIO ();
use Apache::RequestUtil ();

use Apache::Const -compile => qw(DECLINED OK);

use constant METHOD        => 'EMAIL';
use constant SMTP_HOSTNAME => "localhost";

sub handler {
    my $r = shift;

    return Apache::DECLINED unless $r->method eq METHOD;

    Apache::method_register($r->pool, METHOD);
    $r->handler("perl-script");
    $r->push_handlers(PerlResponseHandler => \&send_email_handler);

    return Apache::OK;
}

sub send_email_handler {
    my $r = shift;

    my %headers = map {$_ => $r->headers_in->get($_)} qw(To From Subject);
    my $content = content($r);

    my $status = send_email(\%headers, \$content);

    $r->content_type('text/plain');
    $r->print($status ? "ACK" : "NACK");
    return Apache::OK;
}


sub content {
    my $r = shift;

    $r->setup_client_block;
    return '' unless $r->should_client_block;
    my $len = $r->headers_in->get('content-length');
    my $buf;
    $r->get_client_block($buf, $len);
```

```
    return $buf;
}

sub send_email {
    my($rh_headers, $r_body) = @_;

    require MIME::Lite;
    MIME::Lite->send("smtp", SMTP_HOSTNAME, Timeout => 60);

    my $msg = MIME::Lite->new(%$rh_headers, Data => $$r_body);
    #warn $msg->as_string;
    $msg->send;
}

1;
```

Let's get the less interesting code out of the way. The function content() grabs the request body. The function send_email() sends the email over SMTP. You should adjust the constant `SMTP_HOSTNAME` to point to your outgoing SMTP server. You can replace this function with your own if you prefer to use a different method to send email.

Now to the more interesting functions. The function `handler()` returns immediately and passes the control to the next handler if the request method is not equal to `EMAIL` (set in the `METHOD` constant):

```
return Apache::DECLINED unless $r->method eq METHOD;
```

Next it tells Apache that this new method is a valid one and that the `perl-script` handler will do the processing. Finally it pushes the function `send_email_handler()` to the `PerlResponseHandler` list of handlers:

```
Apache::method_register($r->pool, METHOD);
$r->handler("perl-script");
$r->push_handlers(PerlResponseHandler => \&send_email_handler);
```

The function terminates the header_parser phase by:

```
return Apache::OK;
```

All other phases run as usual, so you can reuse any HTTP protocol hooks, such as authentication and fixup phases.

When the response phase starts `send_email_handler()` is invoked, assuming that no other response handlers were inserted before it. The response handler consists of three parts. Retrieve the email headers `To`, `From` and `Subject`, and the body of the message:

```
my %headers = map {$_ => $r->headers_in->get($_)} qw(To From Subject);
my $content = $r->content;
```

Then send the email:

```
my $status = send_email(\%headers, \$content);
```

Finally return to the client a simple response acknowledging that email has been sent and finish the response phase by returning `Apache::OK`:

```
$r->content_type('text/plain');
$r->print($status ? "ACK" : "NACK");
return Apache::OK;
```

Of course you will want to add extra validations if you want to use this code in production. This is just a proof of concept implementation.

As already mentioned when you extend an HTTP protocol you need to have a client that knows how to use the extension. So here is a simple client that uses `LWP::UserAgent` to issue an `EMAIL` method request over HTTP protocol:

```
file:send_http_email.pl
-----------------------
#!/usr/bin/perl

use strict;
use warnings;

require LWP::UserAgent;

my $url = "http://localhost:8000/email/";

my %headers = (
    From    => 'example@example.com',
    To      => 'example@example.com',
    Subject => '3 weeks in Tibet',
);

my $content = <<EOI;
I didn't have an email software,
but could use HTTP so I'm sending it over HTTP
EOI

my $headers = HTTP::Headers->new(%headers);
my $req = HTTP::Request->new("EMAIL", $url, $headers, $content);
my $res = LWP::UserAgent->new->request($req);
print $res->is_success ? $res->content : "failed";
```

most of the code is just a custom data. The code that does something consists of four lines at the very end. Create `HTTP::Headers` and `HTTP::Request` object. Issue the request and get the response. Finally print the response's content if it was successful or just *"failed"* if not.

Now save the client code in the file *send_http_email.pl*, adjust the *To* field, make the file executable and execute it, after you have restarted the server. You should receive an email shortly to the address set in the *To* field.

1.2.5 PerlInitHandler

When configured inside any container directive, except `<VirtualHost>`, this handler is an alias for `PerlHeaderParserHandler` described earlier. Otherwise it acts as an alias for `PerlPostReadRequestHandler` described earlier.

It is the first handler to be invoked when serving a request.

This phase is of type `RUN_ALL`.

The best example here would be to use `Apache::Reload` which takes the benefit of this directive. Usually `Apache::Reload` is configured as:

```
PerlInitHandler Apache::Reload
PerlSetVar ReloadAll Off
PerlSetVar ReloadModules "MyApache::*"
```

which during the current HTTP request will monitor and reload all `MyApache::*` modules that have been modified since the last HTTP request. However if we move the global configuration into a `<Location>` container:

```
<Location /devel>
    PerlInitHandler Apache::Reload
    PerlSetVar ReloadAll Off
    PerlSetVar ReloadModules "MyApache::*"
    SetHandler perl-script
    PerlResponseHandler ModPerl::Registry
    Options +ExecCGI
</Location>
```

`Apache::Reload` will reload the modified modules, only when a request to the */devel* namespace is issued, because `PerlInitHandler` plays the role of `PerlHeaderParserHandler` here.

1.2.6 PerlAccessHandler

The *access_checker* phase is the first of three handlers that are involved in what's known as AAA: Authentication and Authorization, and Access control.

This phase can be used to restrict access from a certain IP address, time of the day or any other rule not connected to the user's identity.

This phase is of type `RUN_ALL`.

The handler's configuration scope is `DIR`.

The concept behind access checker handler is very simple, return `Apache::FORBIDDEN` if the access is not allowed, otherwise return `Apache::OK`.

The following example handler denies requests made from IPs on the blacklist.

```
file:MyApache/BlockByIP.pm
--------------------------
package MyApache::BlockByIP;

use strict;
use warnings;

use Apache::RequestRec ();
use Apache::Connection ();

use Apache::Const -compile => qw(FORBIDDEN OK);

my %bad_ips = map {$_ => 1} qw(127.0.0.1 10.0.0.4);

sub handler {
    my $r = shift;

    return exists $bad_ips{$r->connection->remote_ip}
        ? Apache::FORBIDDEN
        : Apache::OK;
}

1;
```

The handler retrieves the connection's IP address, looks it up in the hash of blacklisted IPs and forbids the access if found. If the IP is not blacklisted, the handler returns control to the next access checker handler, which may still block the access based on a different rule.

To enable the handler simply add it to the container that needs to be protected. For example to protect an access to the registry scripts executed from the base location */perl* add:

```
<Location /perl/>
    SetHandler perl-script
    PerlResponseHandler ModPerl::Registry
    PerlAccessHandler MyApache::BlockByIP
    Options +ExecCGI
</Location>
```

It's important to notice that `PerlAccessHandler` can be configured for any subsection of the site, no matter whether it's served by a mod_perl response handler or not. For example to run the handler from our example for all requests to the server simply add to *httpd.conf*:

```
<Location />
    PerlAccessHandler MyApache::BlockByIP
</Location>
```

1.2.7 PerlAuthenHandler

The *check_user_id* (*authen*) phase is called whenever the requested file or directory is password protected. This, in turn, requires that the directory be associated with `AuthName`, `AuthType` and at least one `require` directive.

This phase is usually used to verify a user's identification credentials. If the credentials are verified to be correct, the handler should return `Apache::OK`. Otherwise the handler returns `Apache::HTTP_UNAUTHORIZED` to indicate that the user has not authenticated successfully. When Apache sends the HTTP header with this code, the browser will normally pop up a dialog box that prompts the user for login information.

This phase is of type `RUN_FIRST`.

The handler's configuration scope is `DIR`.

The following handler authenticates users by asking for a username and a password and lets them in only if the length of a string made from the supplied username and password and a single space equals to the secret length, specified by the constant `SECRET_LENGTH`.

```
file:MyApache/SecretLengthAuth.pm
---------------------------------
package MyApache::SecretLengthAuth;

use strict;
use warnings;

use Apache::Access ();
use Apache::RequestUtil ();

use Apache::Const -compile => qw(OK DECLINED HTTP_UNAUTHORIZED);

use Apache::Access();

use constant SECRET_LENGTH => 14;

sub handler {
    my $r = shift;

    my ($status, $password) = $r->get_basic_auth_pw;
    return $status unless $status == Apache::OK;

    return Apache::OK
        if SECRET_LENGTH == length join " ", $r->user, $password;

    $r->note_basic_auth_failure;
    return Apache::HTTP_UNAUTHORIZED;
}

1;
```

First the handler retrieves the status of the authentication and the password in plain text. The status will be set to `Apache::OK` only when the user has supplied the username and the password credentials. If the status is different, we just let Apache handle this situation for us, which will usually challenge the client so it'll supply the credentials.

Note that `get_basic_auth_pw()` does a few things behind the scenes, which are important to understand if you plan on implementing your own authentication mechanism that does not use `get_basic_auth_pw()`. First, is checks the value of the configured `AuthType` for the request,

making sure it is `Basic`. Then it makes sure that the Authorization (or Proxy-Authorization) header is formatted for `Basic` authentication. Finally, after isolating the user and password from the header, it populates the *ap_auth_type* slot in the request record with `Basic`. For the first and last parts of this process, mod_perl offers an API. `$r->auth_type` returns the configured authentication type for the current request - whatever was set via the `AuthType` configuration directive. `$r->ap_auth_type` populates the *ap_auth_type* slot in the request record, which should be done after it has been confirmed that the request is indeed using `Basic` authentication. (Note: `$r->ap_auth_type` was `$r->connection->auth_type` in the mod_perl 1.0 API.)

Once we know that we have the username and the password supplied by the client, we can proceed with the authentication. Our authentication algorithm is unusual. Instead of validating the username/password pair against a password file, we simply check that the string built from these two items plus a single space is `SECRET_LENGTH` long (14 in our example). So for example the pair *mod_perl/rules* authenticates correctly, whereas *secret/password* does not, because the latter pair will make a string of 15 characters. Of course this is not a strong authentication scheme and you shouldn't use it for serious things, but it's fun to play with. Most authentication validations simply verify the username/password against a database of valid pairs, usually this requires the password to be encrypted first, since storing passwords in clear is a bad idea.

Finally if our authentication fails the handler calls note_basic_auth_failure() and returns `Apache::HTTP_UNAUTHORIZED`, which sets the proper HTTP response headers that tell the client that its user that the authentication has failed and the credentials should be supplied again.

It's not enough to enable this handler for the authentication to work. You have to tell Apache what authentication scheme to use (`Basic` or `Digest`), which is specified by the `AuthType` directive, and you should also supply the `AuthName` -- the authentication realm, which is really just a string that the client usually uses as a title in the pop-up box, where the username and the password are inserted. Finally the `Require` directive is needed to specify which usernames are allowed to authenticate. If you set it to `valid-user` any username will do.

Here is the whole configuration section that requires users to authenticate before they are allowed to run the registry scripts from */perl/*:

```
<Location /perl/>
    SetHandler perl-script
    PerlResponseHandler ModPerl::Registry
    PerlAuthenHandler MyApache::SecretLengthAuth
    Options +ExecCGI

    AuthType Basic
    AuthName "The Gate"
    Require valid-user
</Location>
```

Just like `PerlAccessHandler` and other mod_perl handlers, `PerlAuthenHandler` can be configured for any subsection of the site, no matter whether it's served by a mod_perl response handler or not. For example to use the authentication handler from the last example for any requests to the site, simply use:

```
<Location />
    PerlAuthenHandler MyApache::SecretLengthAuth
    AuthType Basic
    AuthName "The Gate"
    Require valid-user
</Location>
```

1.2.8 PerlAuthzHandler

The *auth_checker* (*authz*) phase is used for authorization control. This phase requires a successful authentication from the previous phase, because a username is needed in order to decide whether a user is authorized to access the requested resource.

As this phase is tightly connected to the authentication phase, the handlers registered for this phase are only called when the requested resource is password protected, similar to the auth phase. The handler is expected to return `Apache::DECLINED` to defer the decision, `Apache::OK` to indicate its acceptance of the user's authorization, or `Apache::HTTP_UNAUTHORIZED` to indicate that the user is not authorized to access the requested document.

This phase is of type `RUN_FIRST`.

The handler's configuration scope is `DIR`.

Here is the `MyApache::SecretResourceAuthz` handler which grants access to certain resources only to certain users who have already properly authenticated:

```
file:MyApache/SecretResourceAuthz.pm
------------------------------------
package MyApache::SecretResourceAuthz;

use strict;
use warnings;

use Apache::Access ();
use Apache::RequestUtil ();

use Apache::Const -compile => qw(OK HTTP_UNAUTHORIZED);

use Apache::Access ();

my %protected = (
    'admin'  => ['stas'],
    'report' => [qw(stas boss)],
);

sub handler {
    my $r = shift;

    my $user = $r->user;
    if ($user) {
        my($section) = $r->uri =~ m|^/company/(\w+)/|;
        if (defined $section && exists $protected{$section}) {
            my $users = $protected{$section};
```

```
            return Apache::OK if grep { $_ eq $user } @$users;
        }
        else {
            return Apache::OK;
        }
    }

    $r->note_basic_auth_failure;
    return Apache::HTTP_UNAUTHORIZED;
}

1;
```

This authorization handler is very similar to the authentication handler from the previous section. Here we rely on the previous phase to get users authenticated, and now as we have the username we can make decisions whether to let the user access the resource it has asked for or not. In our example we have a simple hash which maps which users are allowed to access what resources. So for example anything under */company/admin/* can be accessed only by the user *stas*, */company/report/* can be accessed by users *stas* and *boss*, whereas any other resources under */company/* can be accessed by everybody who has reached so far. If for some reason we don't get the username, we or the user is not authorized to access the resource the handler does the same thing as it does when the authentication fails, i.e, calls:

```
    $r->note_basic_auth_failure;
    return Apache::HTTP_UNAUTHORIZED;
```

The configuration is similar to the one in the previous section, this time we just add the `PerlAuthzHandler` setting. The rest doesn't change.

```
Alias /company/ /home/httpd/httpd-2.0/perl/
<Location /company/>
    SetHandler perl-script
    PerlResponseHandler ModPerl::Registry
    PerlAuthenHandler MyApache::SecretLengthAuth
    PerlAuthzHandler  MyApache::SecretResourceAuthz
    Options +ExecCGI

    AuthType Basic
    AuthName "The Secret Gate"
    Require valid-user
</Location>
```

And if you want to run the authentication and authorization for the whole site, simply add:

```
<Location />
    PerlAuthenHandler MyApache::SecretLengthAuth
    PerlAuthzHandler  MyApache::SecretResourceAuthz
    AuthType Basic
    AuthName "The Secret Gate"
    Require valid-user
</Location>
```

1.2.9 PerlTypeHandler

The *type_checker* phase is used to set the response MIME type (`Content-type`) and sometimes other bits of document type information like the document language.

For example `mod_autoindex`, which performs automatic directory indexing, uses this phase to map the filename extensions to the corresponding icons which will be later used in the listing of files.

Of course later phases may override the mime type set in this phase.

This phase is of type `RUN_FIRST`.

The handler's configuration scope is `DIR`.

The most important thing to remember when overriding the default *type_checker* handler, which is usually the mod_mime handler, is that you have to set the handler that will take care of the response phase and the response callback function or the code won't work. mod_mime does that based on `SetHandler` and `AddHandler` directives, and file extensions. So if you want the content handler to be run by mod_perl, set either:

```
$r->handler('perl-script');
$r->set_handlers(PerlResponseHandler => \&handler);
```

or:

```
$r->handler('modperl');
$r->set_handlers(PerlResponseHandler => \&handler);
```

depending on which type of response handler is wanted.

Writing a `PerlTypeHandler` handler which sets the content-type value and returns `Apache::DECLINED` so that the default handler will do the rest of the work, is not a good idea, because mod_mime will probably override this and other settings.

Therefore it's the easiest to leave this stage alone and do any desired settings in the *fixups* phase.

1.2.10 PerlFixupHandler

The *fixups* phase is happening just before the content handling phase. It gives the last chance to do things before the response is generated. For example in this phase `mod_env` populates the environment with variables configured with *SetEnv* and *PassEnv* directives.

This phase is of type `RUN_ALL`.

The handler's configuration scope is `DIR`.

The following fixup handler example tells Apache at run time which handler and callback should be used to process the request based on the file extension of the request's URI.

```
file:MyApache/FileExtDispatch.pm
--------------------------------
package MyApache::FileExtDispatch;

use strict;
use warnings;

use Apache::RequestIO ();
use Apache::RequestRec ();

use Apache::Const -compile => 'OK';

use constant HANDLER  => 0;
use constant CALLBACK => 1;

my %exts = (
    cgi => ['perl-script',     \&cgi_handler],
    pl  => ['modperl',         \&pl_handler ],
    tt  => ['perl-script',     \&tt_handler ],
    txt => ['default-handler', undef        ],
);

sub handler {
    my $r = shift;

    my($ext) = $r->uri =~ /\.(\w+)$/;
    $ext = 'txt' unless defined $ext and exists $exts{$ext};

    $r->handler($exts{$ext}->[HANDLER]);

    if (defined $exts{$ext}->[CALLBACK]) {
        $r->set_handlers(PerlResponseHandler => $exts{$ext}->[CALLBACK]);
    }

    return Apache::OK;
}

sub cgi_handler { content_handler($_[0], 'cgi') }
sub pl_handler  { content_handler($_[0], 'pl')  }
sub tt_handler  { content_handler($_[0], 'tt')  }


sub content_handler {
    my($r, $type) = @_;

    $r->content_type('text/plain');
    $r->print("A handler of type '$type' was called");

    return Apache::OK;
}

1;
```

In the example we have used the following mapping.

```
my %exts = (
    cgi => ['perl-script',     \&cgi_handler],
    pl  => ['modperl',         \&pl_handler ],
    tt  => ['perl-script',     \&tt_handler ],
    txt => ['default-handler', undef        ],
);
```

So that *.cgi* requests will be handled by the `perl-script` handler and the `cgi_handler()` callback, *.pl* requests by `modperl` and `pl_handler()`, *.tt* (template toolkit) by `perl-script` and the `tt_handler()`, finally *.txt* request by the `default-handler` handler, which requires no callback.

Moreover the handler assumes that if the request's URI has no file extension or it does, but it's not in its mapping, the `default-handler` will be used, as if the *txt* extension was used.

After doing the mapping, the handler assigns the handler:

```
$r->handler($exts{$ext}->[HANDLER]);
```

and the callback if needed:

```
if (defined $exts{$ext}->[CALLBACK]) {
    $r->set_handlers(PerlResponseHandler => $exts{$ext}->[CALLBACK]);
}
```

In this simple example the callback functions don't do much but calling the same content handler which simply prints the name of the extension if handled by mod_perl, otherwise Apache will serve the other files using the default handler. In real world you will use callbacks to real content handlers that do real things.

Here is how this handler is configured:

```
Alias /dispatch/ /home/httpd/httpd-2.0/htdocs/
<Location /dispatch/>
    PerlFixupHandler MyApache::FileExtDispatch
</Location>
```

Notice that there is no need to specify anything, but the fixup handler. It applies the rest of the settings dynamically at run-time.

1.2.11 PerlResponseHandler

The *handler* (*response*) phase is used for generating the response. This is arguably the most important phase and most of the existing Apache modules do most of their work at this phase.

This is the only phase that requires two directives under mod_perl. For example:

```
<Location /perl>
   SetHandler perl-script
   PerlResponseHandler MyApache::WorldDomination
</Location>
```

`SetHandler` set to `perl-script` or `modperl` tells Apache that mod_perl is going to handle the response generation. `PerlResponseHandler` tells mod_perl which callback is going to do the job.

This phase is of type `RUN_FIRST`.

The handler's configuration scope is `DIR`.

Most of the `Apache::` modules on CPAN are dealing with this phase. In fact most of the developers spend the majority of their time working on handlers that generate response content.

Let's write a simple response handler, that just generates some content. This time let's do something more interesting than printing *"Hello world"*. Let's write a handler that prints itself:

```
file:MyApache/Deparse.pm
------------------------
package MyApache::Deparse;

use strict;
use warnings;

use Apache::RequestRec ();
use Apache::RequestIO ();
use B::Deparse ();

use Apache::Const -compile => 'OK';

sub handler {
    my $r = shift;

    $r->content_type('text/plain');
    $r->print('sub handler ', B::Deparse->new->coderef2text(\&handler));

    return Apache::OK;
}
1;
```

To enable this handler add to *httpd.conf*:

```
<Location /deparse>
    SetHandler modperl
    PerlResponseHandler MyApache::Deparse
</Location>
```

Now when the server is restarted and we issue a request to *http://localhost/deparse* we get the following response:

```
sub handler {
    package MyApache::Deparse;
    my $r = shift @_;
    $r->content_type('text/plain');
    $r->print('sub handler ', 'B::Deparse'->new->coderef2text(\&handler));
    return 0;
}
```

If you compare it to the source code, it's pretty much the same code. `B::Deparse` is fun to play with!

1.2.12 PerlLogHandler

The *log_transaction* phase happens no matter how the previous phases have ended up. If one of the earlier phases has aborted a request, e.g., failed authentication or 404 (file not found) errors, the rest of the phases up to and including the response phases are skipped. But this phase is always executed.

By this phase all the information about the request and the response is known, therefore the logging handlers usually record this information in various ways (e.g., logging to a flat file or a database).

This phase is of type `RUN_ALL`.

The handler's configuration scope is `DIR`.

Imagine a situation where you have to log requests into individual files, one per user. Assuming that all requests start with */users/username/*, so it's easy to categorize requests by the second URI path component. Here is the log handler that does that:

```
file:MyApache/LogPerUser.pm
---------------------------
package MyApache::LogPerUser;

use strict;
use warnings;

use Apache::RequestRec ();
use Apache::Connection ();
use Fcntl qw(:flock);

use Apache::Const -compile => qw(OK DECLINED);

sub handler {
    my $r = shift;

    my($username) = $r->uri =~ m|^/users/([^/]+)|;
    return Apache::DECLINED unless defined $username;

    my $entry = sprintf qq(%s [%s] "%s" %d %d\n),
        $r->connection->remote_ip, scalar(localtime),
        $r->uri, $r->status, $r->bytes_sent;

    my $log_path = Apache::server_root_relative($r->pool,
        "logs/$username.log");
    open my $fh, ">>$log_path" or die "can't open $log_path: $!";
    flock $fh, LOCK_EX;
    print $fh $entry;
    close $fh;

    return Apache::OK;
}
1;
```

First the handler tries to figure out what username the request is issued for, if it fails to match the URI, it simply returns `Apache::DECLINED`, letting other log handlers to do the logging. Though it could return `Apache::OK` since all other log handlers will be run anyway.

Next it builds the log entry, similar to the default *access_log* entry. It's comprised of remote IP, the current time, the uri, the return status and how many bytes were sent to the client as a response body.

Finally the handler appends this entry to the log file for the user the request was issued for. Usually it's safe to append short strings to the file without being afraid of messing up the file, when two files attempt to write at the same time, but just to be on the safe side the handler exclusively locks the file before performing the writing.

To configure the handler simply enable the module with the `PerlLogHandler` directive, inside the wanted section, which was */users/* in our example:

```
<Location /users/>
    SetHandler perl-script
    PerlResponseHandler ModPerl::Registry
    PerlLogHandler MyApache::LogPerUser
    Options +ExecCGI
</Location>
```

After restarting the server and issuing requests to the following URIs:

```
http://localhost/users/stas/test.pl
http://localhost/users/eric/test.pl
http://localhost/users/stas/date.pl
```

The `MyApache::LogPerUser` handler will append to */logs/stas.log*:

```
127.0.0.1 [Sat Aug 31 01:50:38 2002] "/users/stas/test.pl" 200 8
127.0.0.1 [Sat Aug 31 01:50:40 2002] "/users/stas/date.pl" 200 44
```

and to */logs/eric.log*:

```
127.0.0.1 [Sat Aug 31 01:50:39 2002] "/users/eric/test.pl" 200 8
```

It's important to notice that `PerlLogHandler` can be configured for any subsection of the site, no matter whether it's served by a mod_perl response handler or not. For example to run the handler from our example for all requests to the server, simply add to */httpd.conf*:

```
<Location />
    PerlLogHandler MyApache::LogPerUser
</Location>
```

Since the `PerlLogHandler` phase is of type `RUN_ALL`, all other logging handlers will be called as well.

1.2.13 PerlCleanupHandler

There is no *cleanup* Apache phase, it exists only inside mod_perl. It is used to execute some code immediately after the request has been served (the client went away) and before the request object is destroyed.

There are several usages for this use phase. The obvious one is to run a cleanup code, for example removing temporarily created files. The less obvious is to use this phase instead of `PerlLogHandler` if the logging operation is time consuming. This approach allows to free the client as soon as the response is sent.

This phase is of type `RUN_ALL`.

The handler's configuration scope is `DIR`.

There are two ways to register and run cleanup handlers:

1. **Using the `PerlCleanupHandler` phase**

   ```
   PerlCleanupHandler MyApache::Cleanup
   ```

   or:

   ```
   $r->push_handlers(PerlCleanupHandler => \&cleanup);
   ```

   This method is identical to all other handlers.

   In this technique the `cleanup()` callback accepts `$r` as its only argument.

2. **Using `cleanup_register()` acting on the request object's pool**

   Since a request object pool is destroyed at the end of each request, we can register a cleanup callback which will be executed just before the pool is destroyed. For example:

   ```
   $r->pool->cleanup_register(\&cleanup, $arg);
   ```

   The important difference from using the `PerlCleanupHandler` handler, is that here you can pass an optional arbitrary argument to the callback function, and no `$r` argument is passed by default. Therefore if you need to pass any data other than `$r` you may want to use this technique.

Here is an example where the cleanup handler is used to delete a temporary file. The response handler is running `ls -l` and stores the output in temporary file, which is then used by `$r->sendfile` to send the file's contents. We use `push_handlers()` to push `PerlCleanupHandler` to unlink the file at the end of the request.

```
#file:MyApache/Cleanup1.pm
#-------------------------
package MyApache::Cleanup1;

use strict;
use warnings FATAL => 'all';
```

```
use File::Spec::Functions qw(catfile);

use Apache::RequestRec ();
use Apache::RequestIO ();
use Apache::RequestUtil ();

use Apache::Const -compile => qw(OK DECLINED);
use APR::Const    -compile => 'SUCCESS';

my $file = catfile "/tmp", "data";

sub handler {
    my $r = shift;

    $r->content_type('text/plain');

    local @ENV{qw(PATH BASH_ENV)};
    qx(/bin/ls -l > $file);

    my $status = $r->sendfile($file);
    die "sendfile has failed" unless $status == APR::SUCCESS;

    $r->push_handlers(PerlCleanupHandler => \&cleanup);

    return Apache::OK;
}

sub cleanup {
    my $r = shift;

    die "Can't find file: $file" unless -e $file;
    unlink $file or die "failed to unlink $file";

    return Apache::OK;
}
1;
```

Next we add the following configuration:

```
<Location /cleanup1>
    SetHandler modperl
    PerlResponseHandler MyApache::Cleanup1
</Location>
```

Now when a request to */cleanup1* is made, the contents of the current directory will be printed and once the request is over the temporary file is deleted.

This response handler has a problem of running in a multi-process environment, since it uses the same file, and several processes may try to read/write/delete that file at the same time, wrecking havoc. We could have appended the process id $$ to the file's name, but remember that mod_perl 2.0 code may run in the threaded environment, meaning that there will be many threads running in the same process and the $$ trick won't work any longer. Therefore one really has to use this code to create unique, but predictable, file names across threads and processes:

```
sub unique_id {
    require Apache::MPM;
    require APR::OS;
    return Apache::MPM->is_threaded
        ? "$$." . ${ APR::OS::thread_current() }
        : $$;
}
```

In the threaded environment it will return a string containing the process ID, followed by a thread ID. In the non-threaded environment only the process ID will be returned. However since it gives us a predictable string, they may still be a non-satisfactory solution. Therefore we need to use a random string. We can either either Perl's `rand`, some CPAN module or the APR's `APR::UUID`:

```
sub unique_id {
    require APR::UUID;
    return APR::UUID->new->format;
}
```

Now the problem is how do we tell the cleanup handler what file should be cleaned up? We could have stored it in the `$r->notes` table in the response handler and then retrieve it in the cleanup handler. However there is a better way - as mentioned earlier, we can register a callback for request pool cleanup, and when using this method we can pass an arbitrary argument to it. Therefore in our case we choose to pass the file name, based on random string. Here is a better version of the response and cleanup handlers, that uses this technique:

```
#file:MyApache/Cleanup2.pm
#-------------------------
package MyApache::Cleanup2;

use strict;
use warnings FATAL => 'all';

use File::Spec::Functions qw(catfile);

use Apache::RequestRec ();
use Apache::RequestIO ();
use Apache::RequestUtil ();
use APR::UUID ();
use APR::Pool ();

use Apache::Const -compile => qw(OK DECLINED);
use APR::Const    -compile => 'SUCCESS';

my $file_base = catfile "/tmp", "data-";

sub handler {
    my $r = shift;

    $r->content_type('text/plain');
    my $file = $file_base . APR::UUID->new->format;

    local @ENV{qw(PATH BASH_ENV)};
    qx(/bin/ls -l > $file);

    my $status = $r->sendfile($file);
```

```
    die "sendfile has failed" unless $status == APR::SUCCESS;

    $r->pool->cleanup_register(\&cleanup, $file);

    return Apache::OK;
}

sub cleanup {
    my $file = shift;

    die "Can't find file: $file" unless -e $file;
    unlink $file or die "failed to unlink $file";

    return Apache::OK;
}
1;
```

Similarly to the first handler, we add the configuration:

```
<Location /cleanup2>
    SetHandler modperl
    PerlResponseHandler MyApache::Cleanup2
</Location>
```

And now when requesting */cleanup2* we still get the same output -- the listing of the current directory -- but this time this code will work correctly in the multi-processes/multi-threaded environment and temporary files get cleaned up as well.

1.3 Handling HEAD Requests

In order to avoid the overhead of sending the data to the client when the request is of type HEAD in mod_perl 1.0 we used to return early from the handler:

```
return OK if $r->header_only;
```

This logic is no longer needed in mod_perl 2.0, because Apache 2.0 automatically discards the response body for HEAD requests. (You can also read the comment in for `ap_http_header_filter()` in *modules/http/http_protocol.c* in the Apache 2.0 source.)

1.4 Extending HTTP Protocol

Extending HTTP under mod_perl is a trivial task. Look at the example of adding a new method `EMAIL` for details.

1.5 Maintainers

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

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

1.6 Authors

-

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

Table of Contents:

1 HTTP Handlers 1
1.1 Description 2
1.2 HTTP Request Cycle Phases 2
1.2.1 PerlPostReadRequestHandler 4
1.2.2 PerlTransHandler 6
1.2.3 PerlMapToStorageHandler META: add something here 7
1.2.4 PerlHeaderParserHandler 7
1.2.5 PerlInitHandler 11
1.2.6 PerlAccessHandler 11
1.2.7 PerlAuthenHandler 12
1.2.8 PerlAuthzHandler 15
1.2.9 PerlTypeHandler 17
1.2.10 PerlFixupHandler 17
1.2.11 PerlResponseHandler 19
1.2.12 PerlLogHandler 21
1.2.13 PerlCleanupHandler 23
1.3 Handling HEAD Requests 26
1.4 Extending HTTP Protocol 26
1.5 Maintainers 26
1.6 Authors 27