Tutorial Handouts: mod_perl 2.0 By Example

by **Philippe M. Chiasson**
http://gozer.ectoplasm.org/
<gozer@ectoplasm.org>
TicketMaster

ApacheCon US 2004
Saturday, November 13th 2004
Las Vegas, Nevada, USA

This tutorial is available from: http://gozer.ectoplasm.org/talks/

Last modified Sat Nov 13 09:21:54 2004 GMT

1 Getting Your Feet Wet With mod_perl 2.0

1.1 Description

This chapter gives you the bare minimum information to get you started with mod_perl 2.0. For most people it's sufficient to get going.

1.2 Prerequisites

Before building mod_perl 2.0 you need to have its prerequisites installed. If you don't have them, download and install them first, using the information in the following sections. Otherwise proceed directly to the mod_perl building instructions.

The mod_perl 2.0 prerequisites are:

- **Apache**

  Apache 2.0 is required. mod_perl 2.0 **does not** work with Apache 1.3.

- **Perl**
  - **prefork MPM**

    Requires at least Perl version 5.6.0. But we strongly suggest to use at least version 5.6.1, since 5.6.0 is quite buggy.

    You don't need to have threads-support enabled in Perl. If you do have it, it **must** be *ithreads* and not *5005threads*! If you have:

    ```
    % perl5.8.0 -V:use5005threads
    use5005threads='define';
    ```

    you must rebuild Perl without threads enabled or with `-Dusethreads`. Remember that threads support slows things down, so don't enable it unless you really need it.

  - **threaded MPMs**

    Require at least Perl version 5.8.0 with ithreads support built-in. That means that it should report:

    ```
    % perl5.8.0 -V:useithreads -V:usemultiplicity
    useithreads='define';
    usemultiplicity='define';
    ```

    If that's not what you see rebuild Perl with `-Dusethreads`.

1.2.1 Downloading Stable Release Sources

If you are going to install mod_perl on a production site, you want to use the officially released stable components. Since the latest stable versions change all the time you should check for the latest stable version at the listed below URLs:

- **Perl**

  Download from: *http://cpan.org/src/README.html*

  This direct link which symlinks to the latest release should work too: *http://cpan.org/src/stable.tar.gz*.

  For the purpose of examples in this chapter we will use the package named *perl-5.8.x.tar.gz*, where *x* should be replaced with the real version number.

- **Apache**

  Download from: *http://www.apache.org/dist/httpd/*

  For the purpose of examples in this chapter we will use the package named *httpd-2.x.xx.tar.gz*, where *x.xx* should be replaced with the real version number.

1.2.2 Getting Bleeding Edge CVS Sources

If you really know what you are doing you can use the cvs versions of the components. Chances are that you don't want to them on a production site. You have been warned!

- **Perl**

  ```
  # (--delete to ensure a clean state)
  % rsync -acvz --delete --force \
    rsync://ftp.linux.activestate.com/perl-current/ perl-current
  ```

  If you are re-building Perl after rsync-ing, make sure to cleanup first:

  ```
  % make distclean
  ```

  before running `./Configure`.

  You'll also want to install (at least) LWP if you want to fully test mod_perl. You can install LWP with `CPAN.pm` shell:

  ```
  % perl -MCPAN -e 'install("LWP")'
  ```

- **Apache**

  To download the cvs version of httpd-2.0 and bring it to the same state of the distribution package, execute the following commands:

  ```
  % cvs -d :pserver:anoncvs@cvs.apache.org:/home/cvspublic login
  ```

  The password is "anoncvs".

```
% cvs -d :pserver:anoncvs@cvs.apache.org:/home/cvspublic co httpd-2.0
% cd httpd-2.0/srclib
% cvs -d :pserver:anoncvs@cvs.apache.org:/home/cvspublic co apr
% cvs -d :pserver:anoncvs@cvs.apache.org:/home/cvspublic co apr-util
% cd ..
% ./buildconf
```

Once extracted, whenever you want to sync with the latest httpd-2.0 version and rebuild, run:

```
% cd httpd-2.0
% cvs up -dP
% make distclean && ./buildconf
```

1.2.3 Configuring and Installing Prerequisites

If you don’t have the prerequisites installed yet, install them now.

- **Perl**

  ```
  % cd perl-5.8.x
  % ./Configure -des -Dusethreads
  % make && make test && make install
  ```

  If you want to debug mod_perl segmentation faults, add the following *./Configure* options:

  ```
  -Doptimize='-g' -Dusedevel
  ```

- **Apache**

  ```
  % cd httpd-2.x.xx
  % ./configure --prefix=$HOME/httpd/prefork --with-mpm=prefork
  % make && make install
  ```

1.3 mod_perl Installation

First of all check that you have the prerequisites installed.

In this chapter we assume that httpd was installed under *$HOME/httpd/prefork*.

Next, download the mod_perl 2.0 source from: *http://perl.apache.org/download/*.

Now, configure mod_perl:

```
% tar -xvzf mod_perl-2.x.xx.tar.gz
% cd modperl-2.0
% perl Makefile.PL MP_APXS=$HOME/httpd/bin/apxs \
  MP_INST_APACHE2=1
```

where `MP_APXS` is a full path to the `apxs` executable.

Finally, build, test and install mod_perl:

```
  % make && make test && make install
```

Become *root* before doing `make install` if installing system-wide.

If something goes wrong or you need to enable optional features please refer to http://perl.apache.org/docs/2.0/user/install/install.html.

1.4 Configuration

Enable mod_perl built as DSO, by adding to *httpd.conf*:

```
  LoadModule perl_module modules/mod_perl.so
```

Next, tell Perl where to find mod_perl2 libraries:

```
  PerlModule Apache2
```

There are many other configuration options which you can find at
http://perl.apache.org/docs/2.0/user/config/config.html.

If you want to run mod_perl 1.0 code on mod_perl 2.0 server enable the compatibility layer:

```
  PerlModule Apache::compat
```

For more information refer to http://perl.apache.org/docs/2.0/user/compat/compat.html.

1.5 Server Launch and Shutdown

Apache is normally launched with `apachectl`:

```
  % $HOME/httpd/prefork/bin/apachectl start
```

and shut down with:

```
  % $HOME/httpd/prefork/bin/apachectl stop
```

Check *$HOME/httpd/prefork/logs/error_log* to see that the server has started and it's a right one. It should say something similar to:

```
  [Tue May 25 09:24:28 2004] [notice] Apache/2.0.50-dev (Unix)
  mod_perl/1.99_15-dev Perl/v5.8.4 mod_ssl/2.0.50-dev OpenSSL/0.9.7c
  DAV/2 configured -- resuming normal operations
```

1.6 Registry Scripts

To enable registry scripts add to *httpd.conf*:

```
Alias /perl/ /home/httpd/2.0/perl/
<Location /perl/>
    SetHandler perl-script
    PerlResponseHandler ModPerl::Registry
    PerlOptions +ParseHeaders
    Options +ExecCGI
</Location>
```

and now assuming that we have the following script:

```
#!/usr/bin/perl
print "Content-type: text/plain\n\n";
print "mod_perl 2.0 rocks!\n";
```

saved in */home/httpd/httpd-2.0/perl/rock.pl*. Make the script executable and readable by everybody:

```
% chmod a+rx /home/httpd/httpd-2.0/perl/rock.pl
```

Of course the path to the script should be readable by the server too. In the real world you probably want to have a tighter permissions, but for the purpose of testing, that things are working, this is just fine.

Now restart the server and issue a request to *http://localhost/perl/rock.pl* and you should get the response:

```
mod_perl 2.0 rocks!
```

If that didn’t work check the *error_log* file.

1.7 Handler Modules

Finally check that you can run mod_perl handlers. Let’s write a response handler similar to the registry script from the previous section:

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

use strict;
use warnings;

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

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

sub handler {
    my $r = shift;

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

```
      print "mod_perl 2.0 rocks!\n";

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

Save the code in the file *MyApache/Rocks.pm*, somewhere where mod_perl can find it. For example let’s put it under */home/httpd/httpd-2.0/perl/MyApache/Rocks.pm*, and we tell mod_perl that */home/httpd/httpd-2.0/perl/* is in `@INC`, via a startup file which includes just:

```
  use lib qw(/home/httpd/httpd-2.0/perl);
```

and loaded from *httpd.conf*:

```
  PerlRequire /home/httpd/httpd-2.0/perl/startup.pl
```

Now we can configure our module in *httpd.conf*:

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

Now restart the server and issue a request to *http://localhost/rocks* and you should get the response:

```
  mod_perl 2.0 rocks!
```

If that didn’t work check the *error_log* file.

2 New Concepts

2.1 Description

This chapter covers several concepts used in the presentation.

2.2 Exceptions

Apache and APR API return a status code for almost all methods, so if you didn't check the return code and handled any possible problems, you may have silent failures which may cause all kind of obscure problems. On the other hand checking the status code after each call is just too much of a kludge and makes quick prototyping/development almost impossible, not talking about the code readability. Having methods return status codes, also complicates the API if you need to return other values.

Therefore to keep things nice and make the API readable we decided to not return status codes, but instead throw exceptions with `APR::Error` objects for each method that fails. If you don't catch those exceptions, everything works transparently - perl will intercept the exception object and `die()` with a proper error message. So you get all the errors logged without doing any work.

Now, in certain cases you don't want to just die, but instead the error needs to be trapped and handled. For example if some IO operation times out, may be it is OK to trap that and try again. If we were to die with an error message, you would have had to match the error message, which is ugly, inefficient and may not work at all if locale error strings are involved. Therefore you need to be able to get the original status code that Apache or APR has generated. And the exception objects give you that if you want to. Moreover the objects contain additional information, such as the function name (in case you were eval'ing several commands in one block), file and line number where that function was invoked from. More attributes could be added in the future.

`APR::Error` uses Perl operator overloading, such that in boolean and numerical contexts, the object returns the status code; in the string context the full error message is returned.

When intercepting exceptions you need to check whether `$@` is an object (reference). If your application uses other exception objects you additionally need to check whether this is a an `APR::Error` object. Therefore most of the time this is enough:

```
eval { $obj->mp_method() };
if ($@ && $ref $@ && $@ == $some_code)
    warn "handled exception: $@";
}
```

But with other, non-mod_perl, exception objects you need to do:

```
eval { $obj->mp_method() };
if ($@ && $ref $@ eq 'APR::Error' && $@ == $some_code)
    warn "handled exception: $@";
}
```

In theory you could even do:

```
eval { $obj->mp_method() };
if ($@ && $@ == $some_code)
    warn "handled exception: $@";
}
```

but it's possible that the method will die with a plain string and not an object, in which case `$@ == $some_code` won't quite work. Remember that mod_perl throws exception objects only when Apache and APR fail, and in a few other special cases of its own (like *`exit`*).

```
warn "handled exception: $@" if $@ && $ref $@;
```

For example you wrote a code that performs *a socket read*:

```
my $rlen = $sock->recv(my $buff, 1024);
warn "read $rlen bytes\n";
```

and in certain cases it times out. The code will die and log the reason for the failure, which is fine, but later on you may decide that you want to have another attempt to read before dying. In which case you rewrite the code to handle the exception like this:

```
use APR::Const -compile => qw(TIMEUP);
my $buff;
my $tries = 0;
RETRY: my $rlen = eval { $sock->recv($buff, 1024) };
if ($@) {
    die $@ unless ref $@ && $@ == APR::TIMEUP;
    goto RETRY if $tries++ < 3;
}
warn "read $rlen bytes\n";
```

Notice that we handle non-object and non-`APR::Error` exceptions as well, by simply rethrowing them.

You certainly want to have a limit on how many times the code retries the operation as in this example and you probably want to add some fine grained sleep time between attempts, which can be achieved with `select`. As a result the retry code may look like this:

```
RETRY: my $rlen = eval { $sock->recv($buff, 1024) };
if ($@) {
    die $@ unless ref $@ && $@ == APR::TIMEUP;
    if ($tries++ < 3) {
        # sleep 250msec
        select undef, undef, undef, 0.25;
        goto RETRY;
    }
}
```

Finally, the class is called `APR::Error` because it needs to be used outside mod_perl as well, when called from *APR* applications written in Perl.

2.3 Bucket Brigades

Apache 2.0 allows multiple modules to filter both the request and the response. Now one module can pipe its output as an input to another module as if another module was receiving the data directly from the TCP stream. The same mechanism works with the generated response.

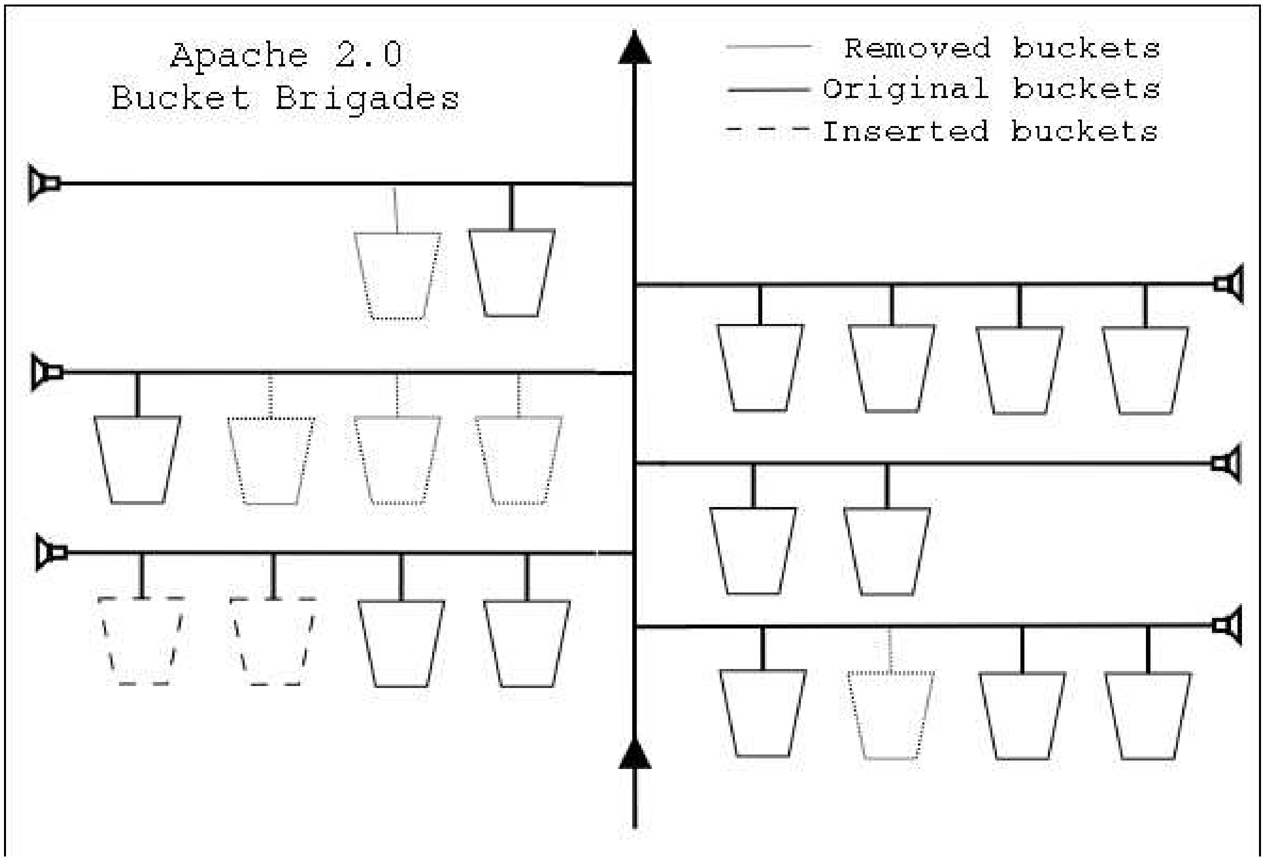
With I/O filtering in place, simple filters, like data compression and decompression, can be easily implemented and complex filters, like SSL, are now possible without needing to modify the the server code which was the case with Apache 1.3.

In order to make the filtering mechanism efficient and avoid unnecessary copying, the *Bucket Brigades* technology was introduced.

A bucket represents a chunk of data. Buckets linked together comprise a brigade. Each bucket in a brigade can be modified, removed and replaced with another bucket. The goal is to minimize the data copying where possible. Buckets come in different types, such as files, data blocks, end of stream indicators, pools, etc. To manipulate a bucket one doesn't need to know its internal representation.

The stream of data is represented by bucket brigades. When a filter is called it gets passed the brigade that was the output of the previous filter. This brigade is then manipulated by the filter (e.g., by modifying some buckets) and passed to the next filter in the stack.

The following figure depicts an imaginary bucket brigade:



The figure tries to show that after the presented bucket brigade has passed through several filters some buckets were removed, some modified and some added. Of course the handler that gets the brigade cannot tell the history of the brigade, it can only see the existing buckets in the brigade.

Bucket brigades are discussed in detail in the protocol handlers and I/O filtering chapters.

3 Introducing mod_perl Handlers

3.1 Description

This chapter provides an introduction into mod_perl handlers.

3.2 Handler Anatomy

Apache distinguishes between numerous phases for which it provides hooks (because the C functions are called *ap_hook_<phase_name>*) where modules can plug various callbacks to extend and alter the default behavior of the webserver. mod_perl provides a Perl interface for most of the available hooks, so mod_perl modules writers can change the Apache behavior in Perl. These callbacks are usually referred to as *handlers* and therefore the configuration directives for the mod_perl handlers look like: `PerlFooHandler`, where `Foo` is one of the handler names. For example `PerlResponseHandler` configures the response callback.

A typical handler is simply a perl package with a *handler* subroutine. For example:

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

use strict;
use warnings;

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

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

sub handler {
    my $r = shift;

    $r->content_type('text/plain');
    $r->print("The time is: " . scalar(localtime) . "\n");

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

This handler simply returns the current date and time as a response.

Since this is a response handler, we configure it as a such in *httpd.conf*:

```
PerlResponseHandler MyApache::CurrentTime
```

Since the response handler should be configured for a specific location, let's write a complete configuration section:

```
PerlModule MyApache::CurrentTime
<Location /time>
    SetHandler modperl
    PerlResponseHandler MyApache::CurrentTime
</Location>
```

Now when a request is issued to *http://localhost/time* this response handler is executed and a response that includes the current time is returned to the client.

3.3 mod_perl Handler Categories

The mod_perl handlers can be divided by their application scope in several categories:

- **Server life cycle**
  - **`PerlOpenLogsHandler`**
  - **`PerlPostConfigHandler`**
  - **`PerlChildInitHandler`**
  - **`PerlChildExitHandler`**
- **Protocols**
  - **`PerlPreConnectionHandler`**
  - **`PerlProcessConnectionHandler`**
- **Filters**
  - **`PerlInputFilterHandler`**
  - **`PerlOutputFilterHandler`**
- **HTTP Protocol**
  - **`PerlPostReadRequestHandler`**
  - **`PerlTransHandler`**
  - **`PerlMapToStorageHandler`**
  - **`PerlInitHandler`**
  - **`PerlHeaderParserHandler`**
  - **`PerlAccessHandler`**
  - **`PerlAuthenHandler`**
  - **`PerlAuthzHandler`**
  - **`PerlTypeHandler`**
  - **`PerlFixupHandler`**
  - **`PerlResponseHandler`**
  - **`PerlLogHandler`**
  - **`PerlCleanupHandler`**

3.4 Stacked Handlers

For each phase there can be more than one handler assigned (also known as *hooks*, because the C functions are called *ap_hook_<phase_name>*). Phases' behavior varies when there is more then one handler registered to run for the same phase. The following table specifies each handler's behavior in this situation:

```
  Directive                   Type
--------------------------------------
PerlOpenLogsHandler          RUN_ALL
PerlPostConfigHandler        RUN_ALL
PerlChildInitHandler         VOID
```

```
PerlChildExitHandler         RUN_ALL

PerlPreConnectionHandler     RUN_ALL
PerlProcessConnectionHandler RUN_FIRST

PerlPostReadRequestHandler   RUN_ALL
PerlTransHandler             RUN_FIRST
PerlMapToStorageHandler      RUN_FIRST
PerlInitHandler              RUN_ALL
PerlHeaderParserHandler      RUN_ALL
PerlAccessHandler            RUN_ALL
PerlAuthenHandler            RUN_FIRST
PerlAuthzHandler             RUN_FIRST
PerlTypeHandler              RUN_FIRST
PerlFixupHandler             RUN_ALL
PerlResponseHandler          RUN_FIRST
PerlLogHandler               RUN_ALL
PerlCleanupHandler           RUN_ALL

PerlInputFilterHandler       VOID
PerlOutputFilterHandler      VOID
```

Note: `PerlChildExitHandler` and `PerlCleanupHandler` are not real Apache hooks, but to mod_perl users they behave as all other hooks.

And here is the description of the possible types:

- **VOID**

  Handlers of the type `VOID` will be *all* executed in the order they have been registered disregarding their return values. Though in mod_perl they are expected to return `Apache::OK`.

- **RUN_FIRST**

  Handlers of the type `RUN_FIRST` will be executed in the order they have been registered until the first handler that returns something other than `Apache::DECLINED`. If the return value is `Apache::DECLINED`, the next handler in the chain will be run. If the return value is `Apache::OK` the next phase will start. In all other cases the execution will be aborted.

- **RUN_ALL**

  Handlers of the type `RUN_ALL` will be executed in the order they have been registered until the first handler that returns something other than `Apache::OK` or `Apache::DECLINED`.

For C API declarations see *include/ap_config.h*, which includes other types which aren't exposed by the mod_perl handlers.

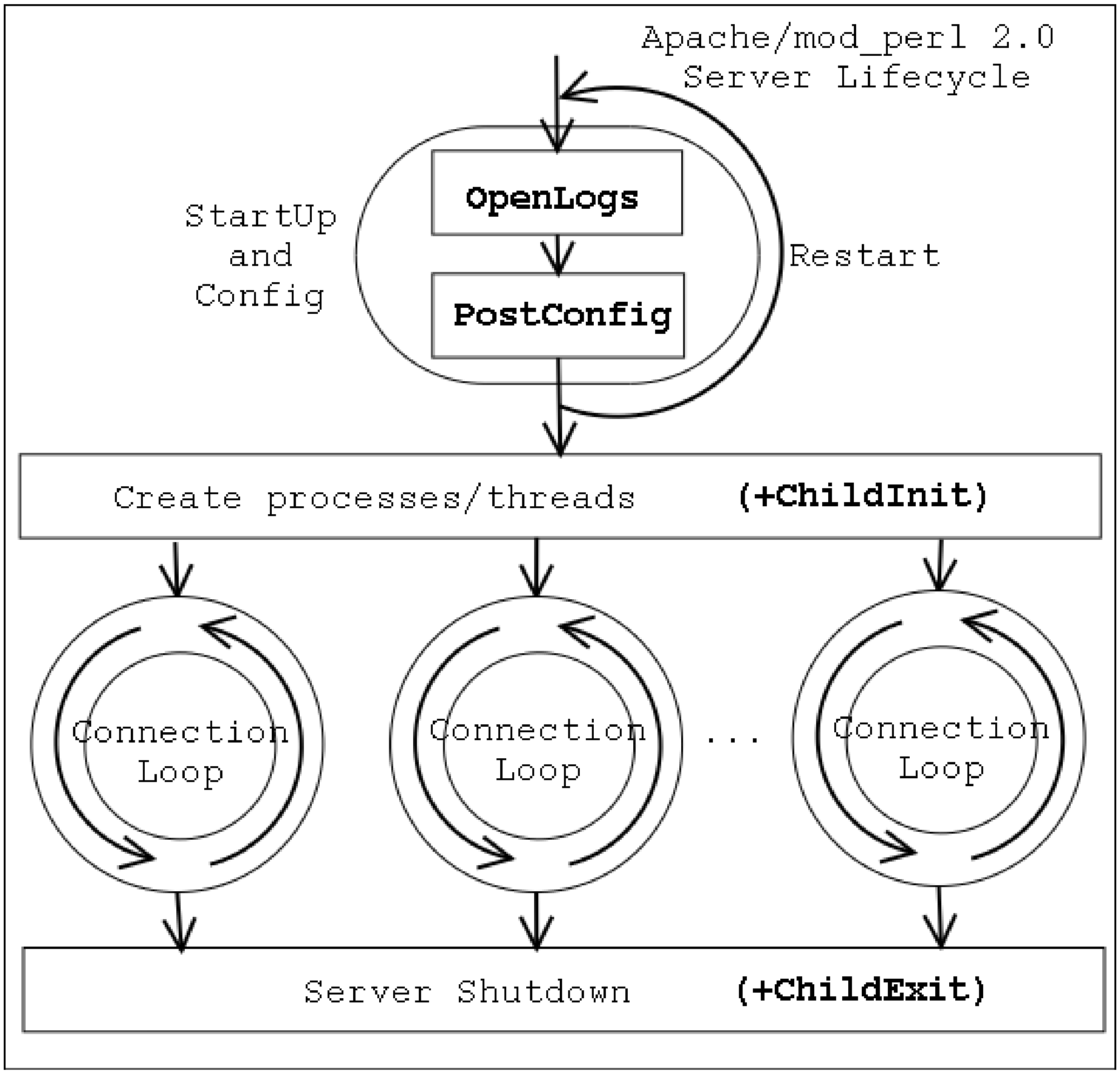
4 Server Life Cycle Handlers

4.1 Description

This chapter discusses server life cycle and the mod_perl handlers participating in it.

4.2 Server Life Cycle

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



Apache 2.0 starts by parsing the configuration file. After the configuration file is parsed, the `PerlOpenLogsHandler` handlers are executed if any. After that it's a turn of `PerlPostConfigHandler` handlers to be run. When the *post_config* phase is finished the server immediately restarts, to make sure

that it can survive graceful restarts after starting to serve the clients.

When the restart is completed, Apache 2.0 spawns the workers that will do the actual work. Depending on the used MPM, these can be threads, processes and a mixture of both. For example the *worker* MPM spawns a number of processes, each running a number of threads. When each child process is started `PerlChildInit` handlers are executed. Notice that they are run for each starting process, not a thread.

From that moment on each working thread processes connections until it’s killed by the server or the server is shutdown.

4.2.1 Startup Phases Demonstration Module

Let’s look at the following example that demonstrates all the startup phases:

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

use strict;
use warnings;

use Apache::Log ();
use Apache::ServerUtil ();

use Fcntl qw(:flock);
use File::Spec::Functions;

use Apache::Const -compile => ’OK’;

my $log_file = catfile "logs", "startup_log";
my $log_fh;

sub open_logs {
    my($conf_pool, $log_pool, $temp_pool, $s) = @_;
    my $log_path = catfile Apache::ServerUtil::server_root, $log_file;

    $s->warn("opening the log file: $log_path");
    open $log_fh, ">>$log_path" or die "can’t open $log_path: $!";
    my $oldfh = select($log_fh); $| = 1; select($oldfh);

    say("process $$ is born to reproduce");
    return Apache::OK;
}

sub post_config {
    my($conf_pool, $log_pool, $temp_pool, $s) = @_;
    say("configuration is completed");
    return Apache::OK;
}

sub child_init {
    my($child_pool, $s) = @_;
    say("process $$ is born to serve");
    return Apache::OK;
```

```
  }

  sub child_exit {
      my($child_pool, $s) = @_;
      say("process $$ now exits");
      return Apache::OK;
  }

  sub say {
      my($caller) = (caller(1))[3] =~ /([^:]+)$/;
      if (defined $log_fh) {
          flock $log_fh, LOCK_EX;
          printf $log_fh "[%s] - %-11s: %s\n",
              scalar(localtime), $caller, $_[0];
          flock $log_fh, LOCK_UN;
      }
      else {
          # when the log file is not open
          warn __PACKAGE__ . " says: $_[0]\n";
      }
  }

  my $parent_pid = $$;
  END {
      my $msg = "process $$ is shutdown";
      $msg .= "\n". "-" x 20 if $$ == $parent_pid;
      say($msg);
  }

  1;
```

And the *httpd.conf* configuration section:

```
  <IfModule prefork.c>
    StartServers         4
    MinSpareServers      4
    MaxSpareServers      4
    MaxClients           10
    MaxRequestsPerChild  0
  </IfModule>

  PerlModule            MyApache::StartupLog
  PerlOpenLogsHandler   MyApache::StartupLog::open_logs
  PerlPostConfigHandler MyApache::StartupLog::post_config
  PerlChildInitHandler  MyApache::StartupLog::child_init
  PerlChildExitHandler  MyApache::StartupLog::child_exit
```

When we perform a server startup followed by a shutdown, the *logs/startup_log* is created if it didn't exist already (it shares the same directory with *error_log* and other standard log files), and each stage appends to that file its log information. So when we perform:

```
  % bin/apachectl start && bin/apachectl stop
```

the following is getting logged to *logs/startup_log*:

```
[Sun Jun  6 01:50:06 2004] - open_logs  : process 24189 is born to reproduce
[Sun Jun  6 01:50:06 2004] - post_config: configuration is completed
[Sun Jun  6 01:50:07 2004] - END        : process 24189 is shutdown
--------------------
[Sun Jun  6 01:50:08 2004] - open_logs  : process 24190 is born to reproduce
[Sun Jun  6 01:50:08 2004] - post_config: configuration is completed
[Sun Jun  6 01:50:09 2004] - child_init : process 24192 is born to serve
[Sun Jun  6 01:50:09 2004] - child_init : process 24193 is born to serve
[Sun Jun  6 01:50:09 2004] - child_init : process 24194 is born to serve
[Sun Jun  6 01:50:09 2004] - child_init : process 24195 is born to serve
[Sun Jun  6 01:50:10 2004] - child_exit : process 24193 now exits
[Sun Jun  6 01:50:10 2004] - END        : process 24193 is shutdown
[Sun Jun  6 01:50:10 2004] - child_exit : process 24194 now exits
[Sun Jun  6 01:50:10 2004] - END        : process 24194 is shutdown
[Sun Jun  6 01:50:10 2004] - child_exit : process 24195 now exits
[Sun Jun  6 01:50:10 2004] - child_exit : process 24192 now exits
[Sun Jun  6 01:50:10 2004] - END        : process 24192 is shutdown
[Sun Jun  6 01:50:10 2004] - END        : process 24195 is shutdown
[Sun Jun  6 01:50:10 2004] - END        : process 24190 is shutdown
--------------------
```

First of all, we can clearly see that Apache always restart itself after the first *post_config* phase is over. The logs show that the *post_config* phase is preceded by the *open_logs* phase. Only after Apache has restarted itself and has completed the *open_logs* and *post_config* phase again, the *child_init* phase is run for each child process. In our example we have had the setting `StartServers=4`, therefore you can see four child processes were started.

Finally you can see that on server shutdown, the *child_exit* phase is run for each child process and the `END {}` block is executed by the parent process and each of the child processes. This is because that `END` block was inherited from the parent on fork.

However the presented behavior varies from MPM to MPM. This demonstration was performed using prefork mpm. Other MPMs like winnt, may run *open_logs* and *post_config* more than once. Also the END blocks may be run more times, when threads are involved. You should be very careful when designing features relying on the phases covered in this chapter if you plan support multiple MPMs. The only thing that’s sure is that you will have each of these phases run at least once.

Apache also specifies the *pre_config* phase, which is executed before the configuration files are parsed, but this is of no use to mod_perl, because mod_perl is loaded only during the configuration phase.

Now let’s discuss each of the mentioned startup handlers and their implementation in the `MyApache::StartupLog` module in detail.

4.2.2 PerlOpenLogsHandler

The *open_logs* phase happens just before the *post_config* phase.

Handlers registered by `PerlOpenLogsHandler` are usually used for opening module-specific log files (e.g., httpd core and mod_ssl open their log files during this phase).

At this stage the `STDERR` stream is not yet redirected to *error_log*, and therefore any messages to that stream will be printed to the console the server is starting from (if such exists).

This phase is of type `RUN_ALL`.

The handler's configuration scope is `SRV`.

As we have seen in the `MyApache::StartupLog::open_logs` handler, the *open_logs* phase handlers accept four arguments: the configuration pool, the logging stream pool, the temporary pool and the main server object:

```
sub open_logs {
    my($conf_pool, $log_pool, $temp_pool, $s) = @_;
    my $log_path = catfile Apache::ServerUtil::server_root, $log_file;

    $s->warn("opening the log file: $log_path");
    open $log_fh, ">>$log_path" or die "can't open $log_path: $!";
    my $oldfh = select($log_fh); $| = 1; select($oldfh);

    say("process $$ is born to reproduce");
    return Apache::OK;
}
```

In our example the handler uses the function `Apache::Server::server_root()` to set the full path to the log file, which is then opened for appending and set to unbuffered mode. Finally it logs the fact that it's running in the parent process.

As you've seen in the example this handler is configured by adding to the top level of *httpd.conf*:

```
PerlOpenLogsHandler MyApache::StartupLog::open_logs
```

This handler can be executed only by the main server. If you want to traverse the configured virtual hosts, you can accomplish that using a simple loop. For example to print out the configured port numbers do:

```
use Apache::Server ();
# ...
sub open_logs {
    my($conf_pool, $log_pool, $temp_pool, $s) = @_;

    my $port = $s->port;
    warn "base port: $port\n";
    for (my $vs = $s->next; $vs; $vs = $vs->next) {
        my $port = $vs->port;
        warn "vhost port: $port\n";
    }
    return Apache::OK;
}
```

`$s` is the base server object.

The pool arguments in this phase and `PerlPostConfigHandler` are:

- `$conf_pool` is the main process sub-pool, therefore its life-span is the same as the main process's one. The main process is a sub-pool of the global pool.
- `$log_pool` is a global pool's sub-pool, therefore its life-span is the same as the Apache program's one.
- `$temp_pool` is a `$conf_pool` subpool, created before the config phase, lives through the open_logs phase and get destroyed after the post_config phase. So you will want to use that pool for doing anything that can be discarded before the requests processing starts.

4.2.3 PerlPostConfigHandler

The *post_config* phase happens right after Apache has processed the configuration files, before any child processes were spawned (which happens at the *child_init* phase).

This phase can be used for initializing things to be shared between all child processes. You can do the same in the startup file, but in the *post_config* phase you have an access to a complete configuration tree, using the `Apache::Directive` module.

This phase is of type `RUN_ALL`.

The handler's configuration scope is `SRV`.

In our `MyApache::StartupLog` example we used the *post_config()* handler:

```
sub post_config {
    my($conf_pool, $log_pool, $temp_pool, $s) = @_;
    say("configuration is completed");
    return Apache::OK;
}
```

As you can see, its arguments are identical to the *open_logs* phase's handler. In this example handler we don't do much but logging that the configuration was completed and returning right away.

As you've seen in the example this handler is configured by adding to *httpd.conf*:

```
PerlPostConfigHandler MyApache::StartupLog::post_config
```

4.2.4 PerlChildInitHandler

The *child_init* phase happens immediately after the child process is spawned. Each child process (not a thread!) will run the hooks of this phase only once in their life-time.

In the prefork MPM this phase is useful for initializing any data structures which should be private to each process. For example `Apache::DBI` pre-opens database connections during this phase and `Apache::Resource` sets the process' resources limits.

This phase is of type `VOID`.

The handler's configuration scope is `SRV`.

In our `MyApache::StartupLog` example we used the *child_init()* handler:

```
sub child_init {
    my($child_pool, $s) = @_;
    say("process $$ is born to serve");
    return Apache::OK;
}
```

The *child_init()* handler accepts two arguments: the child process pool and the server object. The example handler logs the pid of the child process it's run in and returns.

As you've seen in the example this handler is configured by adding to *httpd.conf*:

```
PerlChildInitHandler  MyApache::StartupLog::child_init
```

4.2.5 PerlChildExitHandler

Opposite to the *child_init* phase, the *child_exit* phase is executed before the child process exits. Notice that it happens only when the process exits, not the thread (assuming that you are using a threaded mpm).

This phase is of type *`RUN_ALL`*.

The handler's configuration scope is *`SRV`*.

In our `MyApache::StartupLog` example we used the *child_exit()* handler:

```
sub child_exit {
    my($child_pool, $s) = @_;
    say("process $$ now exits");
    return Apache::OK;
}
```

The *child_exit()* handler accepts two arguments: the child process pool and the server object. The example handler logs the pid of the child process it's run in and returns.

As you've seen in the example this handler is configured by adding to *httpd.conf*:

```
PerlChildExitHandler  MyApache::StartupLog::child_exit
```

5 Protocol Handlers

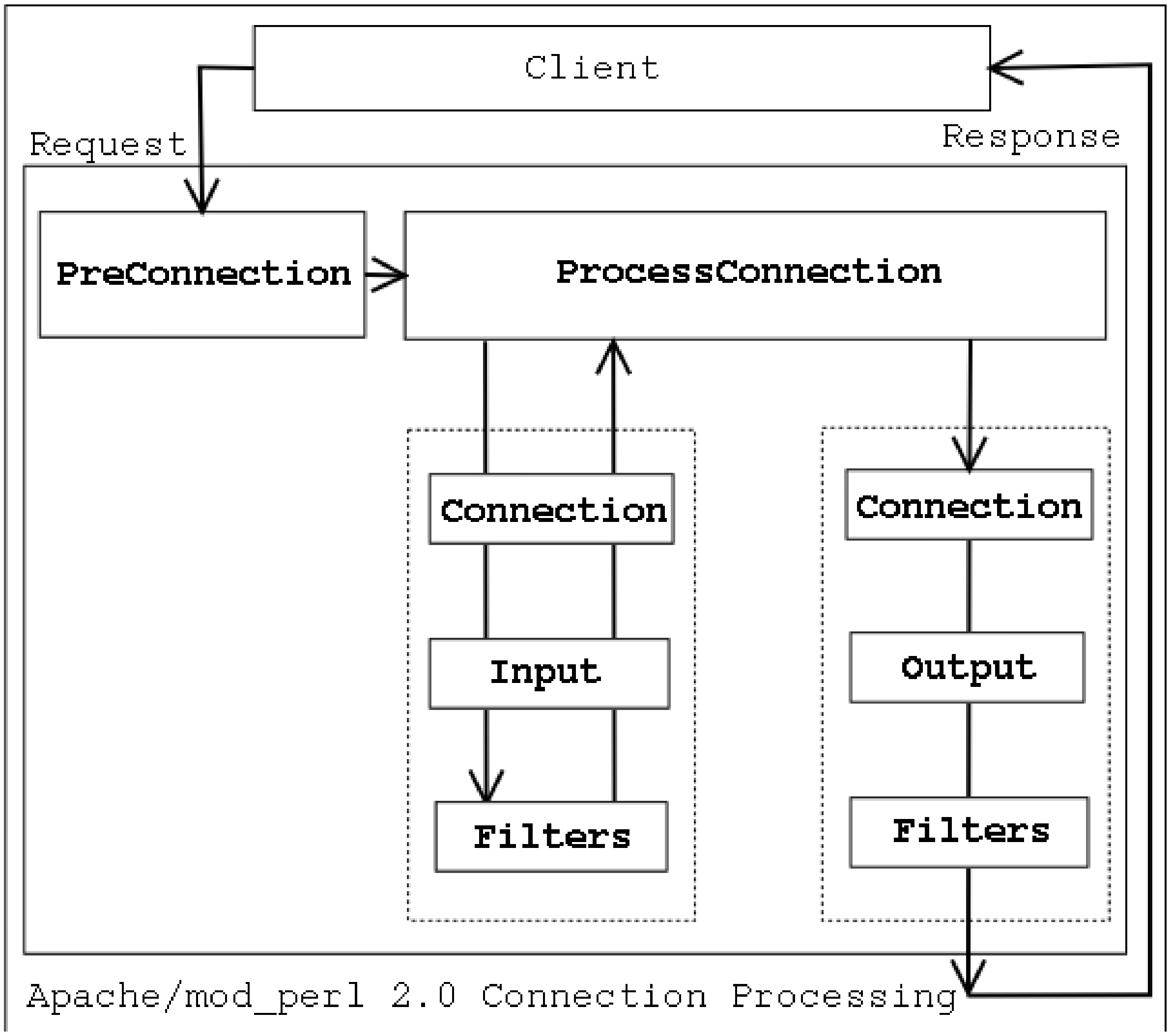
5.1 Description

This chapter explains how to implement Protocol (Connection) Handlers in mod_perl.

5.2 Connection Cycle Phases

As we saw earlier, each child server (be it a thread or a process) is engaged in processing connections. Each connection may be served by different connection protocols, e.g., HTTP, POP3, SMTP, etc. Each connection may include more than one request, e.g., several HTTP requests can be served over a single connection, when several images are requested for the same webpage.

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



When a connection is issued by a client, it's first run through `PerlPreConnectionHandler` and then passed to the `PerlProcessConnectionHandler`, which generates the response. When `PerlProcessConnectionHandler` is reading data from the client, it can be filtered by connection input filters. The generated response can be also filtered though connection output filters. Filters are usually used for modifying the data flowing though them, but can be used for other purposes as well (e.g., logging interesting information).

Now let's discuss each of the `PerlPreConnectionHandler` and `PerlProcessConnectionHandler` handlers in detail.

5.2.1 PerlPreConnectionHandler

The *pre_connection* phase happens just after the server accepts the connection, but before it is handed off to a protocol module to be served. It gives modules an opportunity to modify the connection as soon as possible and insert filters if needed. The core server uses this phase to setup the connection record based on the type of connection that is being used. mod_perl itself uses this phase to register the connection input and output filters.

In mod_perl 1.0 during code development `Apache::Reload` was used to automatically reload modified since the last request Perl modules. It was invoked during *post_read_request*, the first HTTP request's phase. In mod_perl 2.0 *pre_connection* is the earliest phase, so if we want to make sure that all modified Perl modules are reloaded for any protocols and its phases, it's the best to set the scope of the Perl interpreter to the lifetime of the connection via:

```
  PerlInterpScope connection
```

and invoke the `Apache::Reload` handler during the *pre_connection* phase. However this development-time advantage can become a disadvantage in production--for example if a connection, handled by HTTP protocol, is configured as `KeepAlive` and there are several requests coming on the same connection and only one handled by mod_perl and the others by the default images handler, the Perl interpreter won't be available to other threads while the images are being served.

This phase is of type `RUN_ALL`.

The handler's configuration scope is `SRV`, because it's not known yet which resource the request will be mapped to.

A *pre_connection* handler accepts connection record and socket objects as its arguments:

```
  sub handler {
      my($c, $socket) = @_;
      # ...
      return Apache::OK;
  }
```

Here is a useful *pre_connection* phase example: provide a facility to block remote clients by their IP, before too many resources were consumed. This is almost as good as a firewall blocking, as it's executed before Apache has started to do any work at all.

`MyApache::BlockIP2` retrieves client's remote IP and looks it up in the black list (which should certainly live outside the code, e.g. dbm file, but a hardcoded list is good enough for our example).

```
#file:MyApache/BlockIP2.pm
#-------------------------
package MyApache::BlockIP2;

use strict;
use warnings;

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 $c = shift;

    my $ip = $c->remote_ip;
    if (exists $bad_ips{$ip}) {
        warn "IP $ip is blocked\n";
        return Apache::FORBIDDEN;
    }

    return Apache::OK;
}

1;
```

This all happens during the *pre_connection* phase:

```
PerlPreConnectionHandler MyApache::BlockIP2
```

If a client connects from a blacklisted IP, Apache will simply abort the connection without sending any reply to the client, and move on to serving the next request.

5.2.2 PerlProcessConnectionHandler

The *process_connection* phase is used to process incoming connections. Only protocol modules should assign handlers for this phase, as it gives them an opportunity to replace the standard HTTP processing with processing for some other protocols (e.g., POP3, FTP, etc.).

This phase is of type `RUN_FIRST`.

The handler's configuration scope is `SRV`. Therefore the only way to run protocol servers different than the core HTTP is inside dedicated virtual hosts.

A *process_connection* handler accepts a connection record object as its only argument, a socket object can be retrieved from the connection record object.

```
sub handler {
    my ($c) = @_;
    my $socket = $c->client_socket;
    $sock->opt_set(APR::SO_NONBLOCK, 0);
    # ...
    return Apache::OK;
}
```

Most likely you'll need to set the socket to perform blocking IO. On some platforms (e.g. Linux) Apache gives us a socket which is set for blocking, on other platforms (.e.g. Solaris) it doesn't. Unless you know which platforms your application will be running on, always explicitly set it to the blocking IO mode as in the example above. Alternatively, you could query whether the socket is already set to a blocking IO mode with help of the `opt_get()` method.

Now let's look at the following two examples of connection handlers. The first using the connection socket to read and write the data and the second using bucket brigades to accomplish the same and allow for connection filters to do their work.

5.2.2.1 Socket-based Protocol Module

To demonstrate the workings of a protocol module, we'll take a look at the `MyApache::EchoSocket` module, which simply echoes the data read back to the client. In this module we will use the implementation that works directly with the connection socket and therefore bypasses connection filters if any.

A protocol handler is configured using the `PerlProcessConnectionHandler` directive and we will use the `Listen` and `<VirtualHost>` directives to bind to the non-standard port **8010**:

```
Listen 8010
<VirtualHost _default_:8010>
    PerlModule                   MyApache::EchoSocket
    PerlProcessConnectionHandler MyApache::EchoSocket
</VirtualHost>
```

`MyApache::EchoSocket` is then enabled when starting Apache:

```
panic% httpd
```

And we give it a whirl:

```
panic% telnet localhost 8010
Trying 127.0.0.1...
Connected to localhost (127.0.0.1).
Escape character is '^]'.
Hello
Hello

fOo BaR
fOo BaR

Connection closed by foreign host.
```

Here is the code:

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

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

use Apache::Connection ();
use APR::Socket ();

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

use constant BUFF_LEN => 1024;

sub handler {
    my $c = shift;
    my $sock = $c->client_socket;

    # set the socket to the blocking mode
    $sock->opt_set(APR::SO_NONBLOCK => 0);

    while ($sock->recv(my $buff, BUFF_LEN)) {
        last if $buff =~ /^[\r\n]+$/;
        $sock->send($buff);
    }

    Apache::OK;
}
1;
```

The example handler starts with the standard *package* declaration and of course, `use strict;`. As with all `Perl*Handler`s, the subroutine name defaults to *handler*. However, in the case of a protocol handler, the first argument is not a `request_rec`, but a `conn_rec` blessed into the `Apache::Connection` class. We have direct access to the client socket via `Apache::Connection`'s *client_socket* method. This returns an object, blessed into the `APR::Socket` class. Before using the socket, we make sure that it's set to perform blocking IO, by using the `APR::SO_NONBLOCK` constant, compiled earlier.

Inside the recv/send loop, the handler attempts to read `BUFF_LEN` bytes from the client socket into the `$buff` buffer. The handler breaks the loop if nothing was read (EOF) or if the buffer contains nothing but new line character(s), which is how we know to abort the connection in the interactive mode.

If the handler receives some data, it sends it unmodified back to the client with the `APR::Socket::send()` method. When the loop is finished the handler returns `Apache::OK`, telling Apache to terminate the connection. As mentioned earlier since this handler is working directly with the connection socket, no filters can be applied.

5.2.2.2 Bucket Brigades-based Protocol Module

Now let’s look at the same module, but this time implemented by manipulating bucket brigades, and which runs its output through a connection output filter that turns all uppercase characters into their lowercase equivalents.

The following configuration defines a virtual host listening on port 8011 and which enables the `MyApache::EchoBB` connection handler, which will run its output through `MyApache::EchoBB::lowercase_filter` filter:

```
Listen 8011
<VirtualHost _default_:8011>
    PerlModule                   MyApache::EchoBB
    PerlProcessConnectionHandler MyApache::EchoBB
    PerlOutputFilterHandler      MyApache::EchoBB::lowercase_filter
</VirtualHost>
```

As before we start the httpd server:

```
panic% httpd
```

And try the new connection handler in action:

```
panic% telnet localhost 8011
Trying 127.0.0.1...
Connected to localhost (127.0.0.1).
Escape character is ’^]’.
Hello
hello

fOo BaR
foo bar

Connection closed by foreign host.
```

As you can see the response part this time was all in lower case, because of the output filter.

And here is the implementation of the connection and the filter handlers.

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

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

use Apache::Connection ();
use APR::Socket ();
use APR::Bucket ();
use APR::Brigade ();
use APR::Error ();

use APR::Const    -compile => qw(SUCCESS EOF SO_NONBLOCK);
use Apache::Const -compile => qw(OK MODE_GETLINE);
```

```
sub handler {
    my $c = shift;

    $c->client_socket->opt_set(APR::SO_NONBLOCK => 0);

    my $bb_in  = APR::Brigade->new($c->pool, $c->bucket_alloc);
    my $bb_out = APR::Brigade->new($c->pool, $c->bucket_alloc);
    my $ba = $c->bucket_alloc;

    while (1) {
        my $rc = $c->input_filters->get_brigade($bb_in,
                                                Apache::MODE_GETLINE);
        last if $rc == APR::EOF;
        die APR::Error::strerror($rc) unless $rc == APR::SUCCESS;

        while (!$bb_in->is_empty) {
            my $b = $bb_in->first;

            $b->remove;

            if ($b->is_eos) {
                $bb_out->insert_tail($b);
                last;
            }


            if ($b->read(my $data)) {
                last if $data =~ /^[\r\n]+$/;
                # could do some transformation on data here
                $b = APR::Bucket->new($ba, $data);
            }

            $bb_out->insert_tail($b);
        }

        my $fb = APR::Bucket::flush_create($c->bucket_alloc);
        $bb_out->insert_tail($fb);
        $c->output_filters->pass_brigade($bb_out);
    }

    $bb_in->destroy;
    $bb_out->destroy;

    Apache::OK;
}

use base qw(Apache::Filter);
use constant BUFF_LEN => 1024;

sub lowercase_filter : FilterConnectionHandler {
    my $filter = shift;

    while ($filter->read(my $buffer, BUFF_LEN)) {
        $filter->print(lc $buffer);
    }
```

```
      return Apache::OK;
  }

  1;
```

For the purpose of explaining how this connection handler works, we are going to simplify the handler. The whole handler can be represented by the following pseudo-code:

```
  while ($bb_in = get_brigade()) {
      while ($b_in = $bb_in->get_bucket()) {
          $b_in->read(my $data);
          # do something with data
          $b_out = new_bucket($data);

          $bb_out->insert_tail($b_out);
      }
      $bb_out->insert_tail($flush_bucket);
      pass_brigade($bb_out);
  }
```

The handler receives the incoming data via bucket bridges, one at a time in a loop. It then process each bridge, by retrieving the buckets contained in it, reading the data in, then creating new buckets using the received data, and attaching them to the outgoing brigade. When all the buckets from the incoming bucket brigade were transformed and attached to the outgoing bucket brigade, a flush bucket is created and added as the last bucket, so when the outgoing bucket brigade is passed out to the outgoing connection filters, it won't be buffered but sent to the client right away.

It's possible to make the flushing code simpler, by using a dedicated method *`fflush()`* that does just that -- flushing of the bucket brigade. It replaces 3 lines of code:

```
          my $fb = APR::Bucket::flush_create($c->bucket_alloc);
          $bb_out->insert_tail($fb);
          $c->output_filters->pass_brigade($bb_out);
```

with just one line:

```
          $c->output_filters->fflush($bb_out);
```

If you look at the complete handler, the loop is terminated when one of the following conditions occurs: an error happens, the end of stream status code (`APR::EOF`) has been received (no more input at the connection) or when the received data contains nothing but new line characters which we used to to tell the server to terminate the connection.

Notice that this handler could be much simpler, since we don't modify the data. We could simply pass the whole brigade unmodified without even looking at the buckets. But from this example you can see how to write a connection handler where you actually want to read and/or modify the data. To accomplish that modification simply add a code that transforms the data which has been read from the bucket before it's inserted to the outgoing brigade.

We will skip the filter discussion here, since we are going to talk in depth about filters in *the dedicated to filters tutorial*. But all you need to know at this stage is that the data sent from the connection handler is filtered by the outgoing filter and which transforms it to be all lowercase.

And here is the simplified version of this handler, which doesn't attempt to do any transformation, but simply passes the data though:

```
sub handler {
    my $c = shift;

    $c->client_socket->opt_set(APR::SO_NONBLOCK => 0);

    my $bb = APR::Brigade->new($c->pool, $c->bucket_alloc);

    while (1) {
        my $rc = $c->input_filters->get_brigade($bb,
                                                Apache::MODE_GETLINE);
        last if $rc == APR::EOF;
        die APR::Error::strerror($rc) unless $rc == APR::SUCCESS;
        $c->output_filters->fflush($bb);
    }

    $bb->destroy;

    Apache::OK;
}
```

Since the simplified handler no longer has the condition:

```
$last++ if $data =~ /^[\r\n]+$/;
```

which was used to know when to break from the external `while(1)` loop, it will not work in the interactive mode, because when telnet is used we always end the line with `/[\r\n]/`, which will always send data back to the protocol handler and the condition:

```
last if $bb->is_empty;
```

will never be true. However, this latter version works fine when the client is a script and when it stops sending data, our shorter handler breaks out of the loop.

So let's do one more tweak and make the last version work in the interactive telnet mode without manipulating each bucket separately. This time we will use *`flatten()`* to slurp all the data from all the buckets, which saves us the explicit loop over the buckets in the brigade. The handler now becomes:

```
sub handler {
    my $c = shift;

    $c->client_socket->opt_set(APR::SO_NONBLOCK => 0);

    my $bb = APR::Brigade->new($c->pool, $c->bucket_alloc);
    my $ba = $c->bucket_alloc;

    while (1) {
        my $rc = $c->input_filters->get_brigade($bb,
                                                Apache::MODE_GETLINE);
        last if $rc == APR::EOF;
        die APR::Error::strerror($rc) unless $rc == APR::SUCCESS;

        next unless $bb->flatten(my $data);
        $bb->cleanup;
        last if $data =~ /^[\r\n]+$/;

        # could transform data here
        my $b = APR::Bucket->new($ba, $data);
        $bb->insert_tail($b);

        $c->output_filters->fflush($bb);
    }

    $bb->destroy;

    Apache::OK;
}
```

Notice, that once we slurped the data in the buckets, we had to strip the brigade of its buckets, since we re-used the same brigade to send the data out. We used *`cleanup()`* to get rid of the buckets.

6 Input and Output Filters

6.1 Description

This chapter discusses mod_perl's input and output filter handlers.

6.2 Your First Filter

You certainly already know how filters work. That's because you encounter filters so often in real life. There are many places in our lives where filters are used. The purpose of all filters is to apply some transformation to what's coming into the filter, letting something different out of the filter. Certainly in some cases it's possible to modify the source itself, but that makes things unflexible, and but most of the time we have no control over the source. The advantage of using filters to modify something is that they can be replaced when requirements change Filters also can be stacked, which allows us to make each filter do simple transformations. For example by combining several different filters, we can apply multiple transformations. In certain situations combining several filters of the same kind let's us achieve a better quality output.

The mod_perl filters are not any different, they receive some data, modify it and send it out. In the case of filtering the output of the response handler, we could certainly change the response handler's logic to do something different, since we control the response handler. But this may make the code unnecessary complex. If we can apply transformations to the response handler's output, it certainly gives us more flexibility and simplifies things. For example if a response needs to be compressed before sent out, it'd be very inconvenient and inefficient to code in the response handler itself. Using a filter for that purpose is a perfect solution. Similarly, in certain cases, using an input filter to transform the incoming request data is the most wise solution. Think of the same example of having the incoming data coming compressed.

Just like with real life filters, you can pipe several filters to modify each other's output. You can also customize a selection of different filters at run time.

Without much further ado, let's write a simple but useful obfuscation filter for our HTML documents.

We are going to use a very simple obfuscation -- turn an HTML document into a one liner, which will make it harder to read its source without a special processing. To accomplish that we are going to remove characters \012 (`\n`) and \015 (`\r`), which depending on the platform alone or as a combination represent the end of line and a carriage return.

And here is the filter handler code:

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

use strict;
use warnings;

use Apache::Filter ();
use Apache::RequestRec ();
use APR::Table ();
```

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

  use constant BUFF_LEN => 1024;

  sub handler {
      my $f = shift;

      unless ($f->ctx) {
          $f->r->headers_out->unset('Content-Length');
          $f->ctx(1);
      }

      while ($f->read(my $buffer, BUFF_LEN)) {
          $buffer =~ s/[\r\n]//g;
          $f->print($buffer);
      }

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

Next we configure Apache to apply the `MyApache::FilterObfuscate` filter to all requests that get mapped to files with an *".html"* extension:

```
  <Files ~ "\.html">
      PerlOutputFilterHandler MyApache::FilterObfuscate
  </Files>
```

Filter handlers are similar to HTTP handlers, they are expected to return `Apache::OK` or `Apache::DECLINED`, but instead of receiving `$r` (the request object) as the first argument, they receive `$f` (the filter object).

The filter starts by unsetting of the `Content-Length` response header, because it modifies the length of the response body (shrinks it). If the response handler had set the `Content-Length` header and the filter hasn't unset it, the client may have problems receiving the response since it'd expect more data than it was sent.

The core of this filter is a read-modify-print expression in a while loop. The logic is very simple: read at most `BUFF_LEN` characters of data into `$buffer`, apply the regex to remove any occurences of `\n` and `\r` in it, and print the resulting data out. The input data may come from a response handler, or from an upstream filter. The output data goes to the next filter in the output chain. Even though in this example we haven't configured any more filters, internally Apache by itself uses several core filters to manipulate the data and send it out to the client.

As we are going to explain in great detail in the next sections, the same filter may be called many times during a single request, every time receiving a chunk of data. For example if the POSTed request data is 64k long, an input filter could be invoked 8 times, each time receiving 8k of data. The same may happen during response phase, where an upstream filter may split 64k output in 8 8k chunks. The while loop that we just saw is going to read each of these 8k in 8 calls, since it requests 1k on every `read()` call.

Since it’s enough to unset the `Content-Length` header when the filter is called the first time, we need to have some flag telling us whether we have done the job. The method `ctx()` provides this functionality:

```
unless ($f->ctx) {
    $f->r->headers_out->unset('Content-Length');
    $f->ctx(1);
}
```

the `unset()` call will be made only on the first filter call for each request. Of course you can store any kind of a Perl data structure in `$f->ctx` and retrieve it later in subsequent filter invocations of the same request. We will show plenty of examples using this method in the following sections.

Of course the `MyApache::FilterObfuscate` filter logic should take into account situations where removing new line characters will break the correct rendering, as is the case if there are multi-line `<pre>`...`</pre>` entries, but since it escalates the complexity of the filter, we will disregard this requirement for now.

A positive side effect of this obfuscation algorithm is in shortening the amount of the data sent to the client. If you want to look at the production ready implementation, which takes into account the HTML markup specifics, the `Apache::Clean` module, available from CPAN, does just that.

mod_perl I/O filtering follows the Perl’s principle of making simple things easy and difficult things possible. You have seen that it’s trivial to write simple filters. As you read through this tutorial you will see that much more difficult things are possible, even though a more elaborated code will be needed.

6.3 I/O Filtering Concepts

Before introducing the APIs, mod_perl provides for Apache Filtering, there are several important concepts to understand.

6.3.1 Two Methods for Manipulating Data

Apache 2.0 considers all incoming and outgoing data as chunks of information, disregarding their kind and source or storage methods. These data chunks are stored in *buckets*, which form bucket brigades. Input and output filters massage the data in *bucket brigades*. Response and protocol handlers also receive and send data using bucket brigades, though in most cases this is hidden behind wrappers, such as `read()` and `print()`.

mod_perl 2.0 filters can directly manipulate the bucket brigades or use the simplified streaming interface where the filter object acts similar to a filehandle, which can be read from and printed to.

Even though you don’t use bucket brigades directly when you use the streaming filter interface (which works on bucket brigades behind the scenes), it’s still important to understand bucket brigades. For example you need to know that an output filter will be invoked as many times as the number of bucket brigades sent from an upstream filter or a content handler. Or you need to know that the end of stream indicator (EOS) is sometimes sent in a separate bucket brigade, so it shouldn’t be a surprise that the filter was invoked even though no real data went through. As we delve into the filter details you will see that understanding bucket brigades, will help to understand how filters work.

Moreover you will need to understand bucket brigades if you plan to implement protocol modules.

6.3.2 HTTP Request Versus Connection Filters

HTTP request filters are applied when Apache serves an HTTP request.

HTTP request input filters get invoked on the body of the HTTP request only if the body is consumed by the content handler. HTTP request headers are not passed through the HTTP request input filters.

HTTP response output filters get invoked on the body of the HTTP response if the content handler has generated one. HTTP response headers are not passed through the HTTP response output filters.

Connection level filters are applied at the connection level.

A connection may be configured to serve one or more HTTP requests, or handle other protocols. Connection filters see all the incoming and outgoing data. If an HTTP request is served, connection filters can modify the HTTP headers and the body of request and response. If a different protocol is served over connection (e.g. IMAP), the data could have a completely different pattern, than the HTTP protocol (headers + body).

Apache supports several other filter types, which mod_perl 2.0 may support in the future.

6.3.3 Multiple Invocations of Filter Handlers

Unlike other Apache handlers, filter handlers may get invoked more than once during the same request. Filters get invoked as many times as the number of bucket brigades sent from an upstream filter or a content provider.

For example if a content generation handler sends a string, and then forces a flush, following by more data:

```
# assuming buffered STDOUT ($|==0)
$r->print("foo");
$r->rflush;
$r->print("bar");
```

Apache will generate one bucket brigade with two buckets (there are several types of buckets which contain data, one of them is *transient*):

```
bucket type       data
----------------------
1st    transient   foo
2nd    flush
```

and send it to the filter chain. Then assuming that no more data was sent after `print("bar")`, it will create a last bucket brigade containing data:

```
  bucket type       data
  ----------------------
  1st    transient   bar
```

and send it to the filter chain. Finally it'll send yet another bucket brigade with the EOS bucket indicating that there will be no more data sent:

```
  bucket type       data
  ----------------------
  1st    eos
```

The EOS bucket may be attached to the last bucket brigade with the data, rather than be sent in its own brigade, therefore filters should never make an assumption that the EOS bucket is arriving alone in a bucket brigade.

EOS buckets are valid for Request filters. For Connection filters, you will get one only in the response filters only at the end of the connection. See the trick how to workaround this in `Apache::Filter::HTTPHeadersFixup`. Need to mention that in a few other places in this doc.

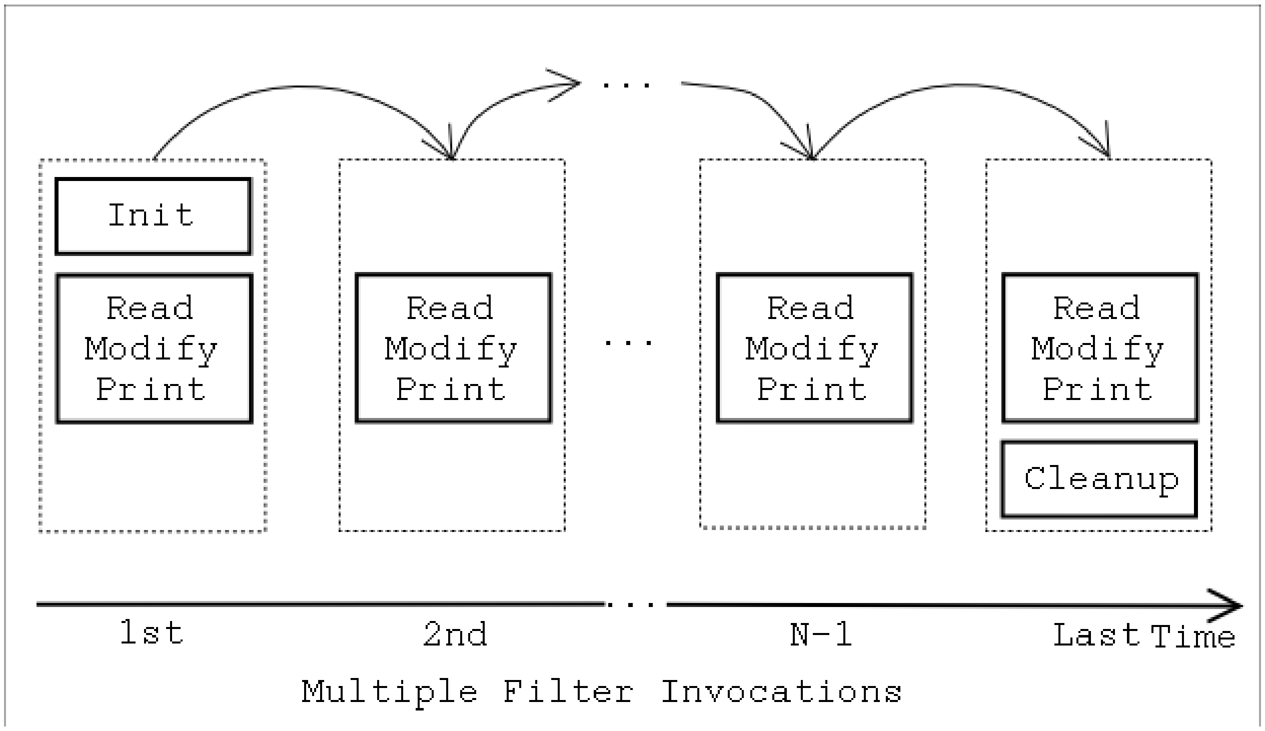
Notice that the EOS bucket may come attached to the last bucket brigade with data, instead of coming in its its own bucket brigade. Filters should never make an assumption that the EOS bucket is arriving alone in a bucket brigade. Therefore the first output filter will be invoked two or three times (three times if EOS is coming in its own brigade), depending on the number of bucket brigades sent by the response handler.

A user may install an upstream filter, and that filter may decide to insert extra bucket brigades or collect all the data in all bucket brigades passing through it and send it all down in one brigade. What's important to remember is when coding a filter, one should never assume that the filter is always going to be invoked once, or a fixed number of times. Neither one can make assumptions on the way the data is going to come in. Therefore a typical filter handler may need to split its logic in three parts.

Jumping ahead we will show some pseudo-code that represents all three parts. This is how a typical stream-oriented filter handler looks like:

```
  sub handler {
      my $f = shift;

      # runs on first invocation
      unless ($f->ctx) {
          init($f);
          $f->ctx(1);
      }

      # runs on all invocations
      process($f);

      # runs on the last invocation
      if ($f->seen_eos) {
          finalize($f);
      }

      return Apache::OK;
  }
  sub init     { ... }
  sub process  { ... }
  sub finalize { ... }
```

The following diagram depicts all three parts:



Let’s explain each part using this pseudo-filter.

1. **Initialization**

   During the initialization, the filter runs all the code that should be performed only once across multiple invocations of the filter (this is during a single request). The filter context is used to accomplish that task. For each new request the filter context is created before the filter is called for the first time and its destroyed at the end of the request.

   ```
   unless ($f->ctx) {
       init($f);
       $f->ctx(1);
   }
   ```

   When the filter is invoked for the first time `$f->ctx` returns `undef` and the custom function init() is called. This function could, for example, retrieve some configuration data, set in *httpd.conf* or initialize some datastructure to its default value.

To make sure that init() won't be called on the following invocations, we must set the filter context before the first invocation is completed:

```
          $f->ctx(1);
```

In practice, the context is not just served as a flag, but used to store real data. For example the following filter handler counts the number of times it was invoked during a single request:

```
  sub handler {
      my $f = shift;

      my $ctx = $f->ctx;
      $ctx->{invoked}++;
      $f->ctx($ctx);
      warn "filter was invoked $ctx->{invoked} times\n";

      return Apache::DECLINED;
  }
```

Since this filter handler doesn't consume the data from the upstream filter, it's important that this handler returns `Apache::DECLINED`, in which case mod_perl passes the current bucket brigade to the next filter. If this handler returns `Apache::OK`, the data will be simply lost. And if that data included a special EOS token, this may wreck havoc.

Unsetting the `Content-Length` header for filters that modify the response body length is a good example of the code to be used in the initialization phase:

```
  unless ($f->ctx) {
      $f->r->headers_out->unset('Content-Length');
      $f->ctx(1);
  }
```

We will see more of initialization examples later in this chapter.

2. **Processing**

The next part:

```
      process($f);
```

is unconditionally invoked on every filter invocation. That's where the incoming data is read, modified and sent out to the next filter in the filter chain. Here is an example that lowers the case of the characters passing through:

```
  use constant READ_SIZE  => 1024;
  sub process {
      my $f = shift;
      while ($f->read(my $data, READ_SIZE)) {
          $f->print(lc $data);
      }
  }
```

Here the filter operates only on a single bucket brigade. Since it manipulates every character separately the logic is really simple.

In more complicated filters the filters may need to buffer data first before the transformation can be applied. For example if the filter operates on html tokens (e.g., ’<img src="../figures/me.jpg">’), it’s possible that one brigade will include the beginning of the token (’<img ’) and the remainder of the token (’src="me.jpg">’) will come in the next bucket brigade (on the next filter invocation). In certain cases it may involve more than two bucket brigades to get the whole token. In such a case the filter will have to store the remainder of unprocessed data in the filter context and then reuse it on the next invocation. Another good example is a filter that performs data compression (compression is usually effective only when applied to relatively big chunks of data), so if a single bucket brigade doesn’t contain enough data, the filter may need to buffer the data in the filter context till it collects enough of it.

We will see the implementation examples in this chapter.

3. **Finalization**

   Finally, some filters need to know when they are invoked for the last time, in order to perform various cleanups and/or flush any remaining data. As mentioned earlier, Apache indicates this event by a special end of stream "token", represented by a bucket of type `EOS`. If the filter is using the streaming interface, rather than manipulating the bucket brigades directly, and it was calling read() in a while loop, it can check whether this is the last time it’s invoked, using the `$f->seen_eos` method:

   ```
   if ($f->seen_eos) {
       finalize($f);
   }
   ```

   This check should be done at the end of the filter handler, because sometimes the EOS "token" comes attached to the tail of data (the last invocation gets both the data and EOS) and sometimes it comes all alone (the last invocation gets only EOS). So if this test is performed at the beginning of the handler and the EOS bucket was sent in together with the data, the EOS event may be missed and filter won’t function properly.

   Jumping ahead, filters, directly manipulating bucket brigades, have to look for a bucket whose type is `EOS` to accomplish this. We will see examples later in the chapter.

Some filters may need to deploy all three parts of the described logic, others will need to do only initialization and processing, or processing and finalization, while the simplest filters might perform only the normal processing (as we saw in the example of the filter handler that lowers the case of the characters going through it).

6.3.4 Blocking Calls

All filters (excluding the core filter that reads from the network and the core filter that writes to it) block at least once when invoked. Depending on whether this is an input or an output filter, the blocking happens when the bucket brigade is requested from the upstream filter or when the bucket brigade is passed to the downstream filter.

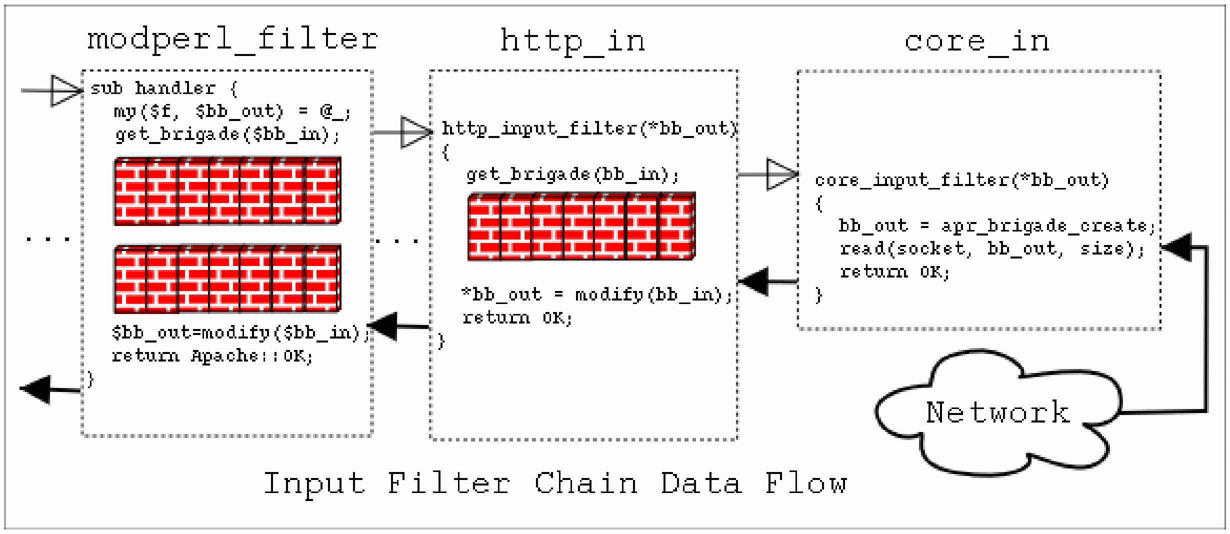
First of all, the input and output filters differ in the ways they acquire the bucket brigades (which includes the data that they filter). Even though when a streaming API is used the difference can't be seen, it's important to understand how things work underneath. Therefore we are going to show examples of transparent filters, which pass data through them unmodified. Instead of reading the data in and printing it out the bucket brigades are now passed as is.

Here is a code for a transparent input filter:

```
#file:MyApache/FilterTransparent.pm (first part)
#-----------------------------------------------
package MyApache::FilterTransparent;

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

sub in {
    my($f, $bb, $mode, $block, $readbytes) = @_;

    my $rv = $f->next->get_brigade($bb, $mode, $block, $readbytes);
    return $rv unless $rv == APR::SUCCESS;

    return Apache::OK;
}
```

When the input filter *in()* is invoked, it first asks the upstream filter for the next bucket brigade (using the `get_brigade()` call). That upstream filter is in turn going to ask for the bucket brigade from the next upstream filter in chain, etc., till the last filter (called `core_in`), that reads from the network is reached. The `core_in` filter reads, using a socket, a portion of the incoming data from the network, processes it and sends it to its downstream filter, which will process the data and send it to its downstream filter, etc., till it reaches the very first filter who has asked for the data. (In reality some other handler triggers the request for the bucket brigade, e.g., an HTTP response handler, or a protocol module, but for our discussion it's good enough to assume that it's the first filter that issues the `get_brigade()` call.)

The following diagram depicts a typical input filters chain data flow in addition to the program control flow.



The black- and white-headed arrows show when the control is switched from one filter to another. In addition the black-headed arrows show the actual data flow. The diagram includes some pseudo-code, both for in Perl for the mod_perl filters and in C for the internal Apache filters. You don't have to understand C to understand this diagram. What's important to understand is that when input filters are invoked they first call each other via the `get_brigade()` call and then block (notice the brick wall on the diagram), waiting for the call to return. When this call returns all upstream filters have already completed finishing their filtering task.

As mentioned earlier, the streaming interface hides these details, however the first `$f->read()` call will block, as underneath it performs the `get_brigade()` call.

The diagram shows a part of the actual input filter chain for an HTTP request, the `...` shows that there are more filters in between the mod_perl filter and `http_in`.

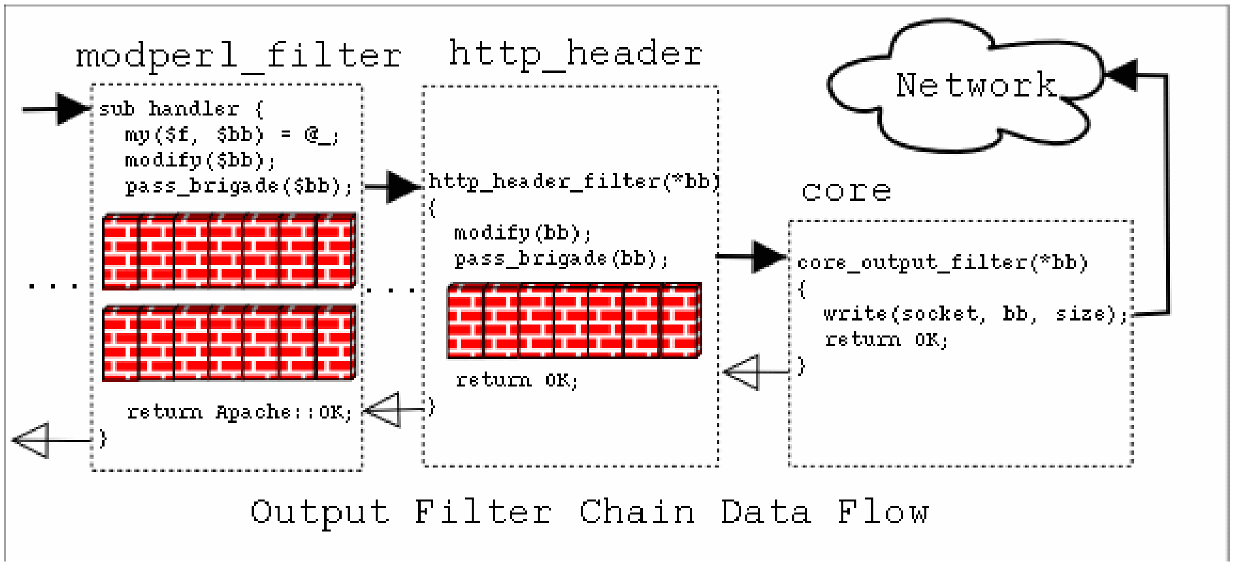
Now let's look at what happens in the output filters chain. Here the first filter acquires the bucket brigades containing the response data, from the content handler (or another protocol handler if we aren't talking HTTP), it then may apply some modification and pass the data to the next filter (using the `pass_brigade()` call), which in turn applies its modifications and sends the bucket brigade to the next filter, etc., all the way down to the last filter (called `core`) which writes the data to the network, via the socket the client is listening to. Even though the output filters don't have to wait to acquire the bucket brigade (since the upstream filter passes it to them as an argument), they still block in a similar fashion to input filters, since they have to wait for the `pass_brigade()` call to return.

Here is an example of a transparent output filter:

```
#file:MyApache/FilterTransparent.pm (continued)
#-----------------------------------------------
sub out {
    my($f, $bb) = @_;

    my $rv = $f->next->pass_brigade($bb);
    return $rv unless $rv == APR::SUCCESS;

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

The *out()* filter passes `$bb` to the downstream filter unmodified and if you add debug prints before and after the `pass_brigade()` call and configure the same filter twice, the debug print will show the blocking call.

The following diagram depicts a typical output filters chain data flow in addition to the program control flow:



Similar to the input filters chain diagram, the arrows show the program control flow and in addition the black-headed arrows show the data flow. Again, it uses a Perl pseudo-code for the mod_perl filter and C pseudo-code for the Apache filters, similarly the brick walls represent the waiting. And again, the diagram shows a part of the real HTTP response filters chain, where `...` stands for the omitted filters.

6.4 mod_perl Filters Declaration and Configuration

Now let’s see how mod_perl filters are declared and configured.

6.4.1 Filter Priority Types

When Apache filters are configured they are inserted into the filters chain according to their priority/type. In most cases when using one or two filters things will just work, however if you find that the order of filter invocation is wrong, the filter priority type should be consulted. Unfortunately this information is available only by consulting the source code, unless it's documented in the module man pages. Numerical definitions of priority types, such as `AP_FTYPE_CONTENT_SET`, `AP_FTYPE_RESOURCE`, can be found in *include/util_filter.h*.

As of this writing Apache comes with two core filters: `DEFLATE` and `INCLUDES`. For example in the following configuration:

```
  SetOutputFilter DEFLATE
  SetOutputFilter INCLUDES
```

the `DEFLATE` filter will be inserted in the filters chain after the `INCLUDES` filter, even though it was configured before it. This is because the `DEFLATE` filter is of type `AP_FTYPE_CONTENT_SET` (20), whereas the `INCLUDES` filter is of type `AP_FTYPE_RESOURCE` (10).

As of this writing mod_perl provides two kind of filters with fixed priority type:

```
  Handler                  Priority           Value
  -------------------------------------------------
  FilterRequestHandler     AP_FTYPE_RESOURCE    10
  FilterConnectionHandler  AP_FTYPE_PROTOCOL    30
```

Therefore `FilterRequestHandler` filters (10) will be always invoked before the `DEFLATE` filter (20), whereas `FilterConnectionHandler` filters (30) after it. The `INCLUDES` filter (10) has the same priority as `FilterRequestHandler` filters (10), and therefore it'll be inserted according to the configuration order, when `PerlSetOutputFilter` or `PerlSetInputFilter` is used.

6.4.2 PerlInputFilterHandler

The `PerlInputFilterHandler` handler registers a filter for input filtering.

This handler is of type `VOID`.

The handler's configuration scope is `DIR`

The following sections include several examples that use the `PerlInputFilterHandler` handler.

6.4.3 PerlOutputFilterHandler

The `PerlOutputFilterHandler` handler registers and configures output filters.

This handler is of type `VOID`.

The handler's configuration scope is `DIR`

The following sections include several examples that use the `PerlOutputFilterHandler` handler.

6.4.4 *`PerlSetInputFilter`*

The `SetInputFilter` directive, documented at *http://httpd.apache.org/docs-2.0/mod/core.html#setinputfilter* sets the filter or filters which will process client requests and POST input when they are received by the server (in addition to any filters configured earlier).

To mix mod_perl and non-mod_perl input filters of the same priority nothing special should be done. For example if we have an imaginary Apache filter `FILTER_FOO` and mod_perl filter `MyApache::FilterInputFoo`, this configuration:

```
SetInputFilter FILTER_FOO
PerlInputFilterHandler MyApache::FilterInputFoo
```

will add both filters, however the order of their invocation might be not the one that you've expected. To make the invocation order the same as the insertion order replace `SetInputFilter` with `PerlSet-InputFilter`, like so:

```
PerlSetInputFilter FILTER_FOO
PerlInputFilterHandler MyApache::FilterInputFoo
```

now `FILTER_FOO` filter will be always executed before the `MyApache::FilterInputFoo` filter, since it was configured before `MyApache::FilterInputFoo` (i.e., it'll apply its transformations on the incoming data last). Here is a diagram input filters chain and the data flow from the network to the response handler for the presented configuration:

```
      response handler
            /\
            ||
        FILTER_FOO
            /\
            ||
  MyApache::FilterInputFoo
            /\
            ||
    core input filters
            /\
            ||
          network
```

As explained in the section Filter Priority Types this directive won't affect filters of different priority. For example assuming that `MyApache::FilterInputFoo` is a `FilterRequestHandler` filter, the configurations:

```
PerlInputFilterHandler MyApache::FilterInputFoo
PerlSetInputFilter DEFLATE
```

and

```
PerlSetInputFilter DEFLATE
PerlInputFilterHandler MyApache::FilterInputFoo
```

are equivalent, because mod_deflate's `DEFLATE` filter has a higher priority than `MyApache::FilterInputFoo`, thefore it'll always be inserted into the filter chain after `MyApache::FilterInputFoo`, (i.e. the `DEFLATE` filter will apply its transformations on the incoming data first). Here is a diagram input filters chain and the data flow from the network to the response handler for the presented configuration:

```
      response handler
             /\
             ||
   MyApache::FilterInputFoo
             /\
             ||
          DEFLATE
             /\
             ||
     core input filters
             /\
             ||
           network
```

`SetInputFilter`'s `;` semantics are supported as well. For example, in the following configuration:

```
PerlInputFilterHandler MyApache::FilterInputFoo
PerlSetInputFilter FILTER_FOO;FILTER_BAR
```

`MyApache::FilterOutputFoo` will be executed first, followed by `FILTER_FOO` and finally by `FILTER_BAR` (again, assuming that all three filters have the same priority).

The `PerlSetInputFilter` directives's configuration scope is `DIR`.

6.4.5 *PerlSetOutputFilter*

The `SetOutputFilter` directive, documented at *http://httpd.apache.org/docs-2.0/mod/core.html#setoutputfilter* sets the filters which will process responses from the server before they are sent to the client (in addition to any filters configured earlier).

To mix mod_perl and non-mod_perl output filters of the same priority nothing special should be done. This configuration:

```
SetOutputFilter INCLUDES
PerlOutputFilterHandler MyApache::FilterOutputFoo
```

will add all two filters to the filter chain, however the order of their invocation might be not the one that you've expected. To preserve the insertion order replace `SetOutputFilter` with `PerlSetOutputFilter`, like so:

```
PerlSetOutputFilter INCLUDES
PerlOutputFilterHandler MyApache::FilterOutputFoo
```

now mod_include's `INCLUDES` filter will be always executed before the `MyApache::FilterOutputFoo` filter. Here is a diagram input filters chain and the data flow from the response handler to the network for the presented configuration:

```
      response handler
             ||
             \/
          INCLUDES
             ||
             \/
   MyApache::FilterOutputFoo
             ||
             \/
     core output filters
             ||
             \/
           network
```

`SetOutputFilter`'s `;` semantics are supported as well. For example, in the following configuration:

```
PerlOutputFilterHandler MyApache::FilterOutputFoo
PerlSetOutputFilter INCLUDES;FILTER_FOO
```

`MyApache::FilterOutputFoo` will be executed first, followed by `INCLUDES` and finally by `FILTER_FOO` (again, assuming that all three filters have the same priority).

Just as explained in the `PerlSetInputFilter` section, if filters have different priorities, the insertion order might be different. For example in the following configuration:

```
PerlSetOutputFilter DEFLATE
PerlSetOutputFilter INCLUDES
PerlOutputFilterHandler MyApache::FilterOutputFoo
```

mod_include's `INCLUDES` filter will be always executed before the `MyApache::FilterOutputFoo` filter. The latter will be followed by mod_deflate's `DEFLATE` filter, even though it was configured before the other two filters. This is because it has a higher priority. And the corresponding diagram looks like so:

```
      response handler
             ||
             \/
          INCLUDES
             ||
             \/
   MyApache::FilterOutputFoo
             ||
             \/
           DEFLATE
             ||
             \/
```

```
     core output filters
             ||
             \/
           network
```

The `PerlSetOutputFilter` directives's configuration scope is `DIR`.

6.4.6 HTTP Request vs. Connection Filters

mod_perl 2.0 supports connection and HTTP request filtering. mod_perl filter handlers specify the type of the filter using the method attributes.

HTTP request filter handlers are declared using the `FilterRequestHandler` attribute. Consider the following request input and output filters skeleton:

```
  package MyApache::FilterRequestFoo;
  use base qw(Apache::Filter);
  
  sub input  : FilterRequestHandler {
      my($f, $bb, $mode, $block, $readbytes) = @_;
      #...
  }
  
  sub output : FilterRequestHandler {
      my($f, $bb) = @_;
      #...
  }
  
  1;
```

If the attribute is not specified, the default `FilterRequestHandler` attribute is assumed. Filters specifying subroutine attributes must subclass `Apache::Filter`, others only need to:

```
  use Apache::Filter ();
```

The request filters are usually configured in the `<Location>` or equivalent sections:

```
  PerlModule MyApache::FilterRequestFoo
  PerlModule MyApache::NiceResponse
  <Location /filter_foo>
      SetHandler modperl
      PerlResponseHandler     MyApache::NiceResponse
      PerlInputFilterHandler  MyApache::FilterRequestFoo::input
      PerlOutputFilterHandler MyApache::FilterRequestFoo::output
  </Location>
```

Now we have the request input and output filters configured.

The connection filter handler uses the `FilterConnectionHandler` attribute. Here is a similar example for the connection input and output filters.

```
package MyApache::FilterConnectionBar;
use base qw(Apache::Filter);

sub input  : FilterConnectionHandler {
    my($f, $bb, $mode, $block, $readbytes) = @_;
    #...
}

sub output : FilterConnectionHandler {
    my($f, $bb) = @_;
    #...
}

1;
```

This time the configuration must be done outside the `<Location>` or equivalent sections, usually within the `<VirtualHost>` or the global server configuration:

```
Listen 8005
<VirtualHost _default_:8005>
    PerlModule MyApache::FilterConnectionBar
    PerlModule MyApache::NiceResponse

    PerlInputFilterHandler  MyApache::FilterConnectionBar::input
    PerlOutputFilterHandler MyApache::FilterConnectionBar::output
    <Location />
        SetHandler modperl
        PerlResponseHandler MyApache::NiceResponse
    </Location>

</VirtualHost>
```

This accomplishes the configuration of the connection input and output filters.

Notice that for HTTP requests the only difference between connection filters and request filters is that the former see everything: the headers and the body, whereas the latter see only the body.

mod_perl provides two interfaces to filtering: a direct bucket brigades manipulation interface and a simpler, stream-oriented interface. The examples in the following sections will help you to understand the difference between the two interfaces.

6.4.7 Filter Initialization Phase

Like in any cool application, there is a hidden door, that let’s you do cool things. mod_perl is not an exception.

where you can plug yet another callback. This *init* callback runs immediately after the filter handler is inserted into the filter chain, before it was invoked for the first time. Here is a skeleton of an init handler:

```
sub init : FilterInitHandler {
    my $f = shift;
    #...
    return Apache::OK;
}
```

The attribute `FilterInitHandler` marks the Perl function suitable to be used as a filter initialization callback, which is called immediately after a filter is inserted to the filter chain and before it's actually called.

For example you may decide to dynamically remove a filter before it had a chance to run, if some condition is true:

```
sub init : FilterInitHandler {
    my $f = shift;
    $f->remove() if should_remove_filter();
    return Apache::OK;
}
```

Not all `Apache::Filter` methods can be used in the init handler, because it's not a filter. Hence you can use methods that operate on the filter itself, such as `remove()` and `ctx()` or retrieve request information, such as `r()` and `c()`. But not methods that operate on data, such as `read()` and `print()`.

In order to hook an init filter handler, the real filter has to assign this callback using the `FilterHasInitHandler` which accepts a reference to the callback function, similar to `push_handlers()`. The used callback function has to have the `FilterInitHandler` attribute. For example:

```
package MyApache::FilterBar;
use base qw(Apache::Filter);
sub init   : FilterInitHandler { ... }
sub filter : FilterRequestHandler FilterHasInitHandler(\&init) {
    my ($f, $bb) = @_;
    # ...
    return Apache::OK;
}
```

While attributes are parsed during the code compilation (it's really a sort of source filter), the argument to the `FilterHasInitHandler()` attribute is compiled at a later stage once the module is compiled.

The argument to `FilterHasInitHandler()` can be any Perl code which when `eval()`'ed returns a code reference. For example:

```
package MyApache::OtherFilter;
use base qw(Apache::Filter);
sub init  : FilterInitHandler { ... }

package MyApache::FilterBar;
use MyApache::OtherFilter;
use base qw(Apache::Filter);
sub get_pre_handler { \&MyApache::OtherFilter::init }
sub filter : FilterHasInitHandler(get_pre_handler()) { ... }
```

Here the `MyApache::FilterBar::filter` handler is configured to run the `MyApache::OtherFilter::init` init handler.

Notice that the argument to `FilterHasInitHandler()` is always `eval()`’ed in the package of the real filter handler (not the init handler). So the above code leads to the following evaluation:

```
$init_sub = eval "package MyApache::FilterBar; get_pre_handler()";
```

though, this is done in C, using the `eval_pv()` C call.

META: currently only one initialization callback can be registered per filter handler. If the need to register more than one arises it should be very easy to extend the functionality.

6.5 All-in-One Filter

Before we delve into the details of how to write filters that do something with the data, lets first write a simple filter that does nothing but snooping on the data that goes through it. We are going to develop the `MyApache::FilterSnoop` handler which can snoop on request and connection filters, in input and output modes.

But first let’s develop a simple response handler that simply dumps the request’s *args* and *content* as strings:

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

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

use Apache::RequestRec ();
use Apache::RequestIO ();
use APR::Brigade ();
use APR::Bucket ();

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

sub handler {
    my $r = shift;
    $r->content_type(’text/plain’);

    $r->print("args:\n", $r->args, "\n");

    if ($r->method_number == Apache::M_POST) {
        my $data = content($r);
        $r->print("content:\n$data\n");
    }

    return Apache::OK;
}

use Apache::Const -compile => qw(MODE_READBYTES);
use APR::Const    -compile => qw(SUCCESS BLOCK_READ);
```

```
  use constant IOBUFSIZE => 8192;

  sub content {
      my $r = shift;

      my $bb = APR::Brigade->new($r->pool, $r->connection->bucket_alloc);

      my $data = '';
      my $seen_eos = 0;

      do {
          $r->input_filters->get_brigade($bb,
              Apache::MODE_READBYTES, APR::BLOCK_READ, IOBUFSIZE);

          while (!$bb->is_empty) {
              my $b = $bb->first;
              $b->remove;

              if ($b->is_eos) {
                  $seen_eos++;
                  last;
              }

              $b->read(my $buf);
              $data .= $buf;
          }

      } while (!$seen_eos);

      $bb->destroy;

      return $data;
  }

  1;
```

which is configured as:

```
  PerlModule MyApache::Dump
  <Location /dump>
      SetHandler modperl
      PerlResponseHandler MyApache::Dump
  </Location>
```

If we issue the following request:

```
  % echo "mod_perl rules" | POST 'http://localhost:8002/dump?foo=1&bar=2'
```

the response will be:

```
  args:
  foo=1&bar=2
  content:
  mod_perl rules
```

As you can see it simply dumped the query string and the posted data.

Now let’s write the snooping filter:

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

use strict;
use warnings;

use base qw(Apache::Filter);
use Apache::FilterRec ();
use APR::Brigade ();
use APR::Bucket ();
use APR::BucketType ();

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

sub connection : FilterConnectionHandler { snoop("connection", @_) }
sub request    : FilterRequestHandler    { snoop("request",    @_) }

sub snoop {
    my $type = shift;
    my($f, $bb, $mode, $block, $readbytes) = @_; # filter args

    # $mode, $block, $readbytes are passed only for input filters
    my $stream = defined $mode ? "input" : "output";

    # read the data and pass-through the bucket brigades unchanged
    if (defined $mode) {
        # input filter
        my $rv = $f->next->get_brigade($bb, $mode, $block, $readbytes);
        return $rv unless $rv == APR::SUCCESS;
        bb_dump($type, $stream, $bb);
    }
    else {
        # output filter
        bb_dump($type, $stream, $bb);
        my $rv = $f->next->pass_brigade($bb);
        return $rv unless $rv == APR::SUCCESS;
    }

    return Apache::OK;
}

sub bb_dump {
    my($type, $stream, $bb) = @_;

    my @data;
    for (my $b = $bb->first; $b; $b = $bb->next($b)) {
        $b->read(my $bdata);
        push @data, $b->type->name, $bdata;
    }

    # send the sniffed info to STDERR so not to interfere with normal
```

```
      # output
      my $direction = $stream eq 'output' ? ">>>" : "<<<";
      print STDERR "\n$direction $type $stream filter\n";

      my $c = 1;
      while (my($btype, $data) = splice @data, 0, 2) {
          print STDERR "    o bucket $c: $btype\n";
          print STDERR "[$data]\n";
          $c++;
      }
  }
  1;
```

This package provides two filter handlers, one for connection and another for request filtering:

```
  sub connection : FilterConnectionHandler { snoop("connection", @_) }
  sub request    : FilterRequestHandler    { snoop("request",    @_) }
```

Both handlers forward their arguments to the `snoop()` function that does the real job. We needed to add these two subroutines in order to assign the two different attributes. Plus the functions pass the filter type to `snoop()` as the first argument, which gets shifted off `@_` and the rest of the `@_` are the arguments that were originally passed to the filter handler.

It's easy to know whether a filter handler is running in the input or the output mode. The arguments `$f` and `$bb` are always passed, whereas the arguments `$mode`, `$block`, and `$readbytes` are passed only to input filter handlers.

If we are in the input mode, in the same call we retrieve the bucket brigade from the previous filter on the input filters stack and immediately link it to the `$bb` variable which makes the bucket brigade available to the next input filter when the filter handler returns. If we forget to perform this linking our filter will become a black hole in which data simply disappears. Next we call `bb_dump()` which dumps the type of the filter and the contents of the bucket brigade to `STDERR`, without influencing the normal data flow.

If we are in the output mode, the `$bb` variable already points to the current bucket brigade. Therefore we can read the contents of the brigade right away. After that we pass the brigade to the next filter.

Let's snoop on connection and request filter levels in both directions by applying the following configuration:

```
  Listen 8008
  <VirtualHost _default_:8008>
      PerlModule MyApache::FilterSnoop
      PerlModule MyApache::Dump

      # Connection filters
      PerlInputFilterHandler  MyApache::FilterSnoop::connection
      PerlOutputFilterHandler MyApache::FilterSnoop::connection

      <Location /dump>
          SetHandler modperl
          PerlResponseHandler MyApache::Dump
          # Request filters
          PerlInputFilterHandler  MyApache::FilterSnoop::request
```

```
        PerlOutputFilterHandler MyApache::FilterSnoop::request
    </Location>

</VirtualHost>
```

Notice that we use a virtual host because we want to install connection filters.

If we issue the following request:

```
% echo "mod_perl rules" | POST 'http://localhost:8008/dump?foo=1&bar=2'
```

We get the same response, when using `MyApache::FilterSnoop`, because our snooping filter didn't change anything. Though there was a lot of output printed to *error_log*. We present it all here, since it helps a lot to understand how filters work.

First we can see the connection input filter at work, as it processes the HTTP headers. We can see that for this request each header is put into a separate brigade with a single bucket. The data is conveniently enclosed by `[]` so you can see the new line characters as well.

```
<<< connection input filter
    o bucket 1: HEAP
[POST /dump?foo=1&bar=2 HTTP/1.1
]

<<< connection input filter
    o bucket 1: HEAP
[TE: deflate,gzip;q=0.3
]

<<< connection input filter
    o bucket 1: HEAP
[Connection: TE, close
]

<<< connection input filter
    o bucket 1: HEAP
[Host: localhost:8008
]

<<< connection input filter
    o bucket 1: HEAP
[User-Agent: lwp-request/2.01
]

<<< connection input filter
    o bucket 1: HEAP
[Content-Length: 14
]

<<< connection input filter
    o bucket 1: HEAP
[Content-Type: application/x-www-form-urlencoded
]
```

```
<<< connection input filter
    o bucket 1: HEAP
[
]
```

Here the HTTP header has been terminated by a double new line. So far all the buckets were of the *HEAP* type, meaning that they were allocated from the heap memory. Notice that the HTTP request input filters will never see the bucket brigades with HTTP headers, as it has been consumed by the last core connection filter.

The following two entries are generated when `MyApache::Dump::handler` reads the POSTed content:

```
<<< connection input filter
    o bucket 1: HEAP
[mod_perl rules]

<<< request input filter
    o bucket 1: HEAP
[mod_perl rules]
    o bucket 2: EOS
[]
```

as we saw earlier on the diagram, the connection input filter is run before the request input filter. Since our connection input filter was passing the data through unmodified and no other custom connection input filter was configured, the request input filter sees the same data. The last bucket in the brigade received by the request input filter is of type *EOS*, meaning that all the input data from the current request has been received.

Next we can see that `MyApache::Dump::handler` has generated its response. However we can see that only the request output filter gets run at this point:

```
>>> request output filter
    o bucket 1: TRANSIENT
[args:
foo=1&bar=2
content:
mod_perl rules
]
```

This happens because Apache hasn't sent yet the response HTTP headers to the client. The request filter sees a bucket brigade with a single bucket of type *TRANSIENT* which is allocated from the stack memory.

The moment the first bucket brigade of the response body has entered the connection output filters, Apache injects a bucket brigade with the HTTP headers. Therefore we can see that the connection output filter is filtering the brigade with HTTP headers (notice that the request output filters don't see it):

```
>>> connection output filter
    o bucket 1: HEAP
[HTTP/1.1 200 OK
Date: Fri, 04 Jun 2004 09:13:26 GMT
Server: Apache/2.0.50-dev (Unix) mod_perl/1.99_15-dev
Perl/v5.8.4 mod_ssl/2.0.50-dev OpenSSL/0.9.7c DAV/2
Connection: close
Transfer-Encoding: chunked
Content-Type: text/plain; charset=ISO-8859-1

]
```

and followed by the first response body's brigade:

```
>>> connection output filter
    o bucket 1: TRANSIENT
[2b
]
    o bucket 2: TRANSIENT
[args:
foo=1&bar=2
content:
mod_perl rules

]
    o bucket 3: IMMORTAL
[
]
```

If the response is large, the request and connection filters will filter chunks of the response one by one.

Finally, Apache sends a series of the bucket brigades to finish off the response, including the end of stream meta-bucket to tell filters that they shouldn't expect any more data, and flush buckets to flush the data, to make sure that any buffered output is sent to the client:

```
>>> connection output filter
    o bucket 1: IMMORTAL
[0

]
    o bucket 2: EOS
[]

>>> connection output filter
    o bucket 1: FLUSH
[]

>>> connection output filter
    o bucket 1: FLUSH
[]
```

This module helps to understand that each filter handler can be called many time during each request and connection. It's called for each bucket brigade.

Also it’s important to mention that HTTP request input filters are invoked only if there is some POSTed data to read and it’s consumed by a content handler.

6.6 Input Filters

mod_perl supports Connection and HTTP Request input filters:

6.6.1 Connection Input Filters

Let’s say that we want to test how our handlers behave when they are requested as `HEAD` requests, rather than `GET`. We can alter the request headers at the incoming connection level transparently to all handlers.

This example’s filter handler looks for data like:

```
  GET /perl/test.pl HTTP/1.1
```

and turns it into:

```
  HEAD /perl/test.pl HTTP/1.1
```

The following input filter handler does that by directly manipulating the bucket brigades:

```
  file:MyApache/InputFilterGET2HEAD.pm
  -----------------------------------
  package MyApache::InputFilterGET2HEAD;
  
  use strict;
  use warnings;
  
  use base qw(Apache::Filter);
  
  use APR::Brigade ();
  use APR::Bucket ();
  
  use Apache::Const -compile => ’OK’;
  use APR::Const    -compile => ’:common’;
  
  sub handler : FilterConnectionHandler {
      my($f, $bb, $mode, $block, $readbytes) = @_;
  
      return Apache::DECLINED if $f->ctx;
  
      my $rv = $f->next->get_brigade($bb, $mode, $block, $readbytes);
      return $rv unless $rv == APR::SUCCESS;
  
      my $ba = $f->c->bucket_alloc;
  
      for (my $b = $bb->first; $b; $b = $bb->next($b)) {
          $b->read(my $data);
          warn("data: $data\n");
  
          if ($data and $data =~ s|^GET|HEAD|) {
              my $bn = APR::Bucket->new($ba, $data);
```

```
                $b->insert_after($bn);
                $b->remove; # no longer needed
                $f->ctx(1); # flag that that we have done the job
                last;
            }
        }

        Apache::OK;
    }

    1;
```

The filter handler is called for each bucket brigade, which in turn includes buckets with data. The gist of any input filter handler is to request the bucket brigade from the upstream filter, and return it downstream filter using the second argument `$bb`. It's important to remember that you can call methods on this argument, but you shouldn't assign to this argument, or the chain will be broken. You have two techniques to choose from to retrieve-modify-return bucket brigades:

1. Create a new empty bucket brigade `$ctx_bb`, pass it to the upstream filter via `get_brigade()` and wait for this call to return. When it returns, `$ctx_bb` is populated with buckets. Now the filter should move the bucket from `$ctx_bb` to `$bb`, on the way modifying the buckets if needed. Once the buckets are moved, and the filter returns, the downstream filter will receive the populated bucket brigade.

2. Pass `$bb` to `get_brigade()` to the upstream filter, so it will be populated with buckets. Once `get_brigade()` returns, the filter can go through the buckets and modify them in place, or it can do nothing and just return (in which case, the downstream filter will receive the bucket brigade unmodified).

Both techniques allow addition and removal of buckets. Though the second technique is more efficient since it doesn't have the overhead of create the new brigade and moving the bucket from one brigade to another. In this example we have chosen to use the second technique, in the next example we will see the first technique.

Our filter has to perform the substitution of only one HTTP header (which normally resides in one bucket), so we have to make sure that no other data gets mangled (e.g. there could be POSTED data and it may match `/^GET/` in one of the buckets). We use `$f->ctx` as a flag here. When it's undefined the filter knows that it hasn't done the required substitution, though once it completes the job it sets the context to 1.

To optimize the speed, the filter immediately returns `Apache::DECLINED` when it's invoked after the substitution job has been done:

```
return Apache::DECLINED if $f->ctx;
```

In that case mod_perl will call `get_brigade()` internally which will pass the bucket brigade to the downstream filter. Alternatively the filter could do:

```
my $rv = $f->next->get_brigade($bb, $mode, $block, $readbytes);
return $rv unless $rv == APR::SUCCESS;
return Apache::OK if $f->ctx;
```

but this is a bit less efficient.

[META: the most efficient thing to do is to remove the filter itself once the job is done, so it won’t be even invoked after the job has been done.

```
if ($f->ctx) {
    $f->remove;
    return Apache::DECLINED;
}
```

However, this can’t be used with Apache 2.0.49 and lower, since it has a bug when trying to remove the edge connection filter (it doesn’t remove it). Most likely that problem will be not fixed in the 2.0 series due to design flows. I don’t know if it’s going to be fixed in 2.1 series.]

If the job wasn’t done yet, the filter calls `get_brigade`, which populates the `$bb` bucket brigade. Next, the filter steps through the buckets looking for the bucket that matches the regex: `/^GET/`. If that happens, a new bucket is created with the modified data (`s/^GET/HEAD/`. Now it has to be inserted in place of the old bucket. In our example we insert the new bucket after the bucket that we have just modified and immediately remove that bucket that we don’t need anymore:

```
            $b->insert_after($bn);
            $b->remove; # no longer needed
```

Finally we set the context to 1, so we know not to apply the substitution on the following data and break from the *for* loop.

The handler returns `Apache::OK` indicating that everything was fine. The downstream filter will receive the bucket brigade with one bucket modified.

Now let’s check that the handler works properly. For example, consider the following response handler:

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

use strict;
use warnings;

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

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

sub handler {
    my $r = shift;

    $r->content_type('text/plain');
    my $response = "the request type was " . $r->method;
    $r->set_content_length(length $response);
    $r->print($response);
```

```
      Apache::OK;
  }

  1;
```

which returns to the client the request type it has issued. In the case of the `HEAD` request Apache will discard the response body, but it'll will still set the correct `Content-Length` header, which will be 24 in case of the `GET` request and 25 for `HEAD`. Therefore if this response handler is configured as:

```
  Listen 8005
  <VirtualHost _default_:8004>
      <Location />
          SetHandler modperl
          PerlResponseHandler +MyApache::RequestType
      </Location>
  </VirtualHost>
```

and a `GET` request is issued to */*:

```
  panic% perl -MLWP::UserAgent -le \
  '$r = LWP::UserAgent->new()->get("http://localhost:8004/"); \
  print $r->headers->content_length . ": ".  $r->content'
  24: the request type was GET
```

where the response's body is:

```
  the request type was GET
```

And the `Content-Length` header is set to 24.

However if we enable the `MyApache::InputFilterGET2HEAD` input connection filter:

```
  Listen 8005
  <VirtualHost _default_:8005>
      PerlInputFilterHandler +MyApache::InputFilterGET2HEAD

      <Location />
          SetHandler modperl
          PerlResponseHandler +MyApache::RequestType
      </Location>
  </VirtualHost>
```

And issue the same `GET` request, we get only:

```
  25:
```

which means that the body was discarded by Apache, because our filter turned the `GET` request into a `HEAD` request and if Apache wasn't discarding the body on `HEAD`, the response would be:

```
  the request type was HEAD
```

that’s why the content length is reported as 25 and not 24 as in the real GET request.

6.6.2 HTTP Request Input Filters

Request filters are really non-different from connection filters, other than that they are working on request and response bodies and have an access to a request object.

6.6.3 Bucket Brigade-based Input Filters

Let’s look at the request input filter that lowers the case of the request’s body: `MyApache::InputRequestFilterLC`:

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

use strict;
use warnings;

use base qw(Apache::Filter);

use Apache::Connection ();
use APR::Brigade ();
use APR::Bucket ();

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

sub handler : FilterRequestHandler {
    my($f, $bb, $mode, $block, $readbytes) = @_;

    my $bb_ctx = APR::Brigade->new($f->c->pool, $f->c->bucket_alloc);
    my $rv = $f->next->get_brigade($bb_ctx, $mode, $block, $readbytes);
    return $rv unless $rv == APR::SUCCESS;

    my $ba = $f->c->bucket_alloc;

    while (!$bb_ctx->is_empty) {
        my $b = $bb_ctx->first;

        $b->remove;

        if ($b->is_eos) {
            $bb->insert_tail($b);
            last;
        }

        $b->read(my $data);
        $b = APR::Bucket->new($ba, lc $data);

        $bb->insert_tail($b);
    }
```

```
      Apache::OK;
  }

  1;
```

As promised, in this filter handler we have used the first technique of bucket brigade modification. The handler creates a temporary bucket brigade (`ctx_bb`), populates it with data using `get_brigade()`, and then moves buckets from it to the bucket brigade `$bb`, which is then retrieved by the downstream filter when our handler returns.

This filter doesn't need to know whether it was invoked for the first time or whether it has already done something. It's a state-less handler, since it has to lower case everything that passes through it. Notice that this filter can't be used as the connection filter for HTTP requests, since it will invalidate the incoming request headers; for example the first header line:

```
  GET /perl/TEST.pl HTTP/1.1
```

will become:

```
  get /perl/test.pl http/1.1
```

which messes up the request method, the URL and the protocol.

Now if we use the `MyApache::Dump` response handler, we have developed before in this chapter, which dumps the query string and the content body as a response, and configure the server as follows:

```
  <Location /lc_input>
      SetHandler modperl
      PerlResponseHandler    +MyApache::Dump
      PerlInputFilterHandler +MyApache::InputRequestFilterLC
  </Location>
```

When issuing a POST request:

```
 % echo "mOd_pErl RuLeS" | POST 'http://localhost:8002/lc_input?FoO=1&BAR=2'
```

we get a response:

```
  args:
  FoO=1&BAR=2
  content:
  mod_perl rules
```

indeed we can see that our filter has lowercased the POSTed body, before the content handler received it. You can see that the query string wasn't changed.

6.6.4 Stream-oriented Input Filters

Let’s now look at the same filter implemented using the stream-oriented API.

```
file:MyApache/InputRequestFilterLC2.pm
-------------------------------------
package MyApache::InputRequestFilterLC2;

use strict;
use warnings;

use base qw(Apache::Filter);

use Apache::Const -compile => ’OK’;

use constant BUFF_LEN => 1024;

sub handler : FilterRequestHandler {
    my $f = shift;

    while ($f->read(my $buffer, BUFF_LEN)) {
        $f->print(lc $buffer);
    }

    Apache::OK;
}
1;
```

Now you probably ask yourself why did we have to go through the bucket brigades filters when this all can be done so much simpler. The reason is that we wanted you to understand how the filters work underneath, which will assist a lot when you will need to debug filters or optimize their speed. In certain cases a bucket brigade filter may be more efficient than the stream-oriented. For example if the filter applies transformation to selected buckets, certain buckets may contain open filehandles or pipes, rather than real data. And when you call read() the buckets will be forced to read that data in. But if you didn’t want to modify these buckets you could pass them as they are and let Apache do faster techniques for sending data from the file handles or pipes.

The logic is very simple here, the filter reads in loop, and prints the modified data, which at some point will be sent to the next filter. This point happens every time the internal mod_perl buffer is full or when the filter returns.

`read()` populates `$buffer` to a maximum of `BUFF_LEN` characters (1024 in our example). Assuming that the current bucket brigade contains 2050 chars, `read()` will get the first 1024 characters, then 1024 characters more and finally the remaining 2 characters. Notice that even though the response handler may have sent more than 2050 characters, every filter invocation operates on a single bucket brigade so you have to wait for the next invocation to get more input. In one of the earlier examples we have shown that you can force the generation of several bucket brigades in the content handler by using `rflush()`. For example:

```
  $r->print("string");
  $r->rflush();
  $r->print("another string");
```

It’s only possible to get more than one bucket brigade from the same filter handler invocation if the filter is not using the streaming interface and by simply calling `get_brigade()` as many times as needed or till EOS is received.

The configuration section is pretty much identical:

```
  <Location /lc_input2>
      SetHandler modperl
      PerlResponseHandler    +MyApache::Dump
      PerlInputFilterHandler +MyApache::InputRequestFilterLC2
  </Location>
```

When issuing a POST request:

```
 % echo "mOd_pErl RuLeS" | POST ’http://localhost:8002/lc_input2?FoO=1&BAR=2’
```

we get a response:

```
  args:
  FoO=1&BAR=2
  content:
  mod_perl rules
```

indeed we can see that our filter has lowercased the POSTed body, before the content handler received it. You can see that the query string wasn’t changed.

6.7 Output Filters

mod_perl supports Connection and HTTP Request output filters:

6.7.1 Connection Output Filters

Connection filters filter **all** the data that is going through the server. Therefore if the connection is of HTTP request type, connection output filters see the headers and the body of the response, whereas request output filters see only the response body.

6.7.2 HTTP Request Output Filters

As mentioned earlier output filters can be written using the bucket brigades manipulation or the simplified stream-oriented interface.

First let’s develop a response handler that sends two lines of output: numerals 1234567890 and the English alphabet in a single string:

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

use strict;
use warnings;

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

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

sub handler {
    my $r = shift;

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

    $r->print(1..9, "0\n");
    $r->print('a'..'z', "\n");

    Apache::OK;
}
1;
```

The purpose of our filter handler is to reverse every line of the response body, preserving the new line characters in their places. Since we want to reverse characters only in the response body, without breaking the HTTP headers, we will use the HTTP request output filter.

6.7.3 Stream-oriented Output Filters

The first filter implementation is using the stream-oriented filtering API:

```
file:MyApache/FilterReverse1.pm
----------------------------
package MyApache::FilterReverse1;

use strict;
use warnings;

use base qw(Apache::Filter);

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

use constant BUFF_LEN => 1024;

sub handler : FilterRequestHandler {
    my $f = shift;

    while ($f->read(my $buffer, BUFF_LEN)) {
        for (split "\n", $buffer) {
            $f->print(scalar reverse $_);
            $f->print("\n");
        }
    }
```

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

Next, we add the following configuration to *httpd.conf*:

```
PerlModule MyApache::FilterReverse1
PerlModule MyApache::SendAlphaNum
<Location /reverse1>
    SetHandler modperl
    PerlResponseHandler     MyApache::SendAlphaNum
    PerlOutputFilterHandler MyApache::FilterReverse1
</Location>
```

Now when a request to */reverse1* is made, the response handler `MyApache::SendAlphaNum::handler()` sends:

```
1234567890
abcdefghijklmnopqrstuvwxyz
```

as a response and the output filter handler `MyApache::FilterReverse1::handler` reverses the lines, so the client gets:

```
0987654321
zyxwvutsrqponmlkjihgfedcba
```

The `Apache::Filter` module loads the `read()` and `print()` methods which encapsulate the stream-oriented filtering interface.

The reversing filter is quite simple: in the loop it reads the data in the *readline()* mode in chunks up to the buffer length (1024 in our example), and then prints each line reversed while preserving the new line control characters at the end of each line. Behind the scenes `$f->read()` retrieves the incoming brigade and gets the data from it, and `$f->print()` appends to the new brigade which is then sent to the next filter in the stack. `read()` breaks the *while* loop, when the brigade is emptied or the end of stream is received.

In order not to distract the reader from the purpose of the example the used code is oversimplified and won't handle correctly input lines which are longer than 1024 characters and possibly using a different line termination token (could be "\n", "\r" or "\r\n" depending on a platform). Moreover a single line may be split between across two or even more bucket brigades, so we have to store the unprocessed string in the filter context, so it can be used on the following invocations. So here is an example of a more complete handler, which does takes care of these issues:

```
sub handler {
    my $f = shift;

    my $leftover = $f->ctx;
    while ($f->read(my $buffer, BUFF_LEN)) {
        $buffer = $leftover . $buffer if defined $leftover;
        $leftover = undef;
        while ($buffer =~ /([^\r\n]*)([\r\n]*)/g) {
            $leftover = $1, last unless $2;
```

```
                $f->print(scalar(reverse $1), $2);
            }
        }

        if ($f->seen_eos) {
            $f->print(scalar reverse $leftover) if defined $leftover;
        }
        else {
            $f->ctx($leftover) if defined $leftover;
        }

        return Apache::OK;
    }
```

The handler uses the `$leftover` variable to store unprocessed data as long as it fails to assemble a complete line or there is an incomplete line following the new line token. On the next handler invocation this data is then prepended to the next chunk that is read. When the filter is invoked on the last time, it unconditionally reverses and flushes any remaining data.

6.7.4 Bucket Brigade-based Output Filters

The following filter implementation is using the bucket brigades API to accomplish exactly the same task as the first filter.

```
    file:MyApache/FilterReverse2.pm
    --------------------------------
    package MyApache::FilterReverse2;

    use strict;
    use warnings;

    use base qw(Apache::Filter);

    use APR::Brigade ();
    use APR::Bucket ();

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

    sub handler : FilterRequestHandler {
        my($f, $bb) = @_;

        my $bb_ctx = APR::Brigade->new($f->c->pool, $f->c->bucket_alloc);

        my $ba = $f->c->bucket_alloc;

        while (!$bb->is_empty) {
            my $b = $bb->first;

            $b->remove;

            if ($b->is_eos) {
                $bb_ctx->insert_tail($b);
                last;
            }
```

```
              if ($b->read(my $data)) {
                  $data = join "",
                      map {scalar(reverse $_), "\n"} split "\n", $data;
                  $b = APR::Bucket->new($ba, $data);
              }

              $bb_ctx->insert_tail($b);
          }


          my $rv = $f->next->pass_brigade($bb_ctx);
          return $rv unless $rv == APR::SUCCESS;

          Apache::OK;
      }
      1;
```

and the corresponding configuration:

```
  PerlModule MyApache::FilterReverse2
  PerlModule MyApache::SendAlphaNum
  <Location /reverse2>
      SetHandler modperl
      PerlResponseHandler     MyApache::SendAlphaNum
      PerlOutputFilterHandler MyApache::FilterReverse2
  </Location>
```

Now when a request to */reverse2* is made, the client gets:

```
  0987654321
  zyxwvutsrqponmlkjihgfedcba
```

as expected.

The bucket brigades output filter version is just a bit more complicated than the stream-oriented one. The handler receives the incoming bucket brigade `$bb` as its second argument. Since when the handler is completed it must pass a brigade to the next filter in the stack, we create a new bucket brigade into which we are going to put the modified buckets and which eventually we pass to the next filter.

The core of the handler is in removing buckets from the head of the bucket brigade `$bb` while there are some, reading the data from the buckets, reversing and putting it into a newly created bucket which is inserted to the end of the new bucket brigade. If we see a bucket which designates the end of stream, we insert that bucket to the tail of the new bucket brigade and break the loop. Finally we pass the created brigade with modified data to the next filter and return.

Similarly to the original version of `MyApache::FilterReverse1::handler`, this filter is not smart enough to handle incomplete lines. However the exercise of making the filter foolproof should be trivial by porting a better matching rule and using the `$leftover` buffer from the previous section is trivial and left as an exercise to the reader.

7 HTTP Handlers

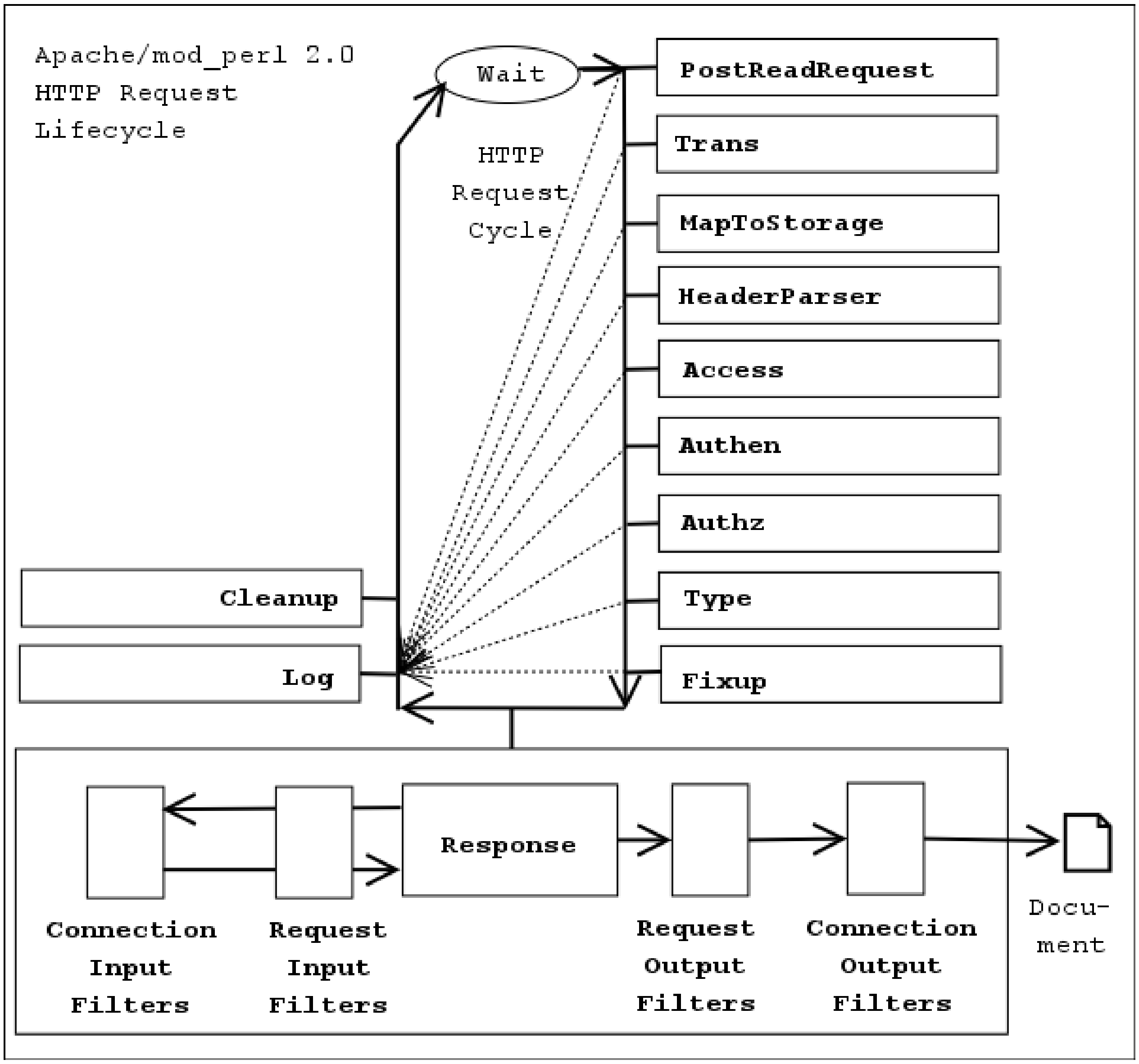
7.1 Description

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

7.2 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 different things are:

- a new directive *`PerlMapToStorageHandler`* was added to match the new phase *map_to_storage* added by Apache 2.0.
- the `PerlHandler` directive has been renamed to `PerlResponseHandler` to better match the corresponding Apache phase name (*response*).
- the *response* phase now includes filtering.



From the diagram it can be seen that an HTTP request is processes by 12 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.

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.

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

7.2.1 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 Apache::RequestIO ();
use File::Spec::Functions qw(catfile);

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

my $conf_file = catfile Apache::ServerUtil::server_root,
    "conf", "httpd.conf";

printf "$conf_file is %0.2f minutes old\n", 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 `PerlPostReadRequestHandler` 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.

7.2.2 PerlTransHandler

The *translate* phase is used to perform the manipulation of a request's URI. 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
```

7.2.3 PerlMapToStorageHandler

The *map_to_storage* phase is used to perform the translation of a request's URI into a corresponding filename. If no custom handler is provided, the server will try to walk the filesystem trying to find what file or directory corresponds to the request's URI. Since usually mod_perl handler don't have corresponding files on the filesystem, you will want to shortcut this phase and save quite a few CPU cycles.

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.

For example if you don't want Apache to try to attempt to translate URI into a filename, just add a handler:

```
PerlMapToStorageHandler MyApache::NoTranslation
```

using the following code:

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

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

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

sub handler {
    my $r = shift;

    # skip ap_directory_walk stat() calls
    return Apache::OK;
}
1;
```

Apache also uses this phase to handle `TRACE` requests. So if you shortcut it, `TRACE` calls will be not handled. In case you need to handle such, you may rewrite it as:

```
file:MyApache/NoTranslation2.pm
-------------------------------
package MyApache::NoTranslation2;

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

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

sub handler {
    my $r = shift;

    return Apache::DECLINED if $r->method_number == Apache::M_TRACE;

    # skip ap_directory_walk stat() calls
    return Apache::OK;
}
1;
```

Another way to prevent the core translation is to set `$r->filename()` to some value, which can also be done in the `PerlTransHandler`, if you are already using it.

7.2.4 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::Server ();
use Apache::Process ();
use APR::Table ();

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::RequestUtil::method_register($r->server->process->pconf,
                                         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 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;
}

use APR::Brigade ();
use APR::Bucket ();

use Apache::Const -compile => qw(MODE_READBYTES);
use APR::Const    -compile => qw(SUCCESS BLOCK_READ);
use constant IOBUFSIZE => 8192;

sub content {
    my $r = shift;

    my $bb = APR::Brigade->new($r->pool, $r->connection->bucket_alloc);

    my $data = '';
    my $seen_eos = 0;
    do {
        $r->input_filters->get_brigade($bb,
            Apache::MODE_READBYTES, APR::BLOCK_READ, IOBUFSIZE);

        while (!$bb->is_empty) {
            my $b = $bb->first;
            $b->remove;

            if ($b->is_eos) {

                $seen_eos++;
                last;
            }

            $b->read(my $buf);
            $data .= $buf;
        }

    } while (!$seen_eos);

    $bb->destroy;

    return $data;
}

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.

```
Apache::RequestUtil::method_register($r->server->process->pconf,
                                     METHOD);
$r->handler("perl-script");
```

Notice that we use the `pconf` pool which persists through the server life, and not `$r->pool` whose life span will end at the end of the request.

Finally it pushes the function `send_email_handler()` to the `PerlResponseHandler` list of handlers:

```
      $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.

7.2.5 PerlInitHandler

When configured inside any container directive, except `<VirtualHost>`, this handler is an alias for `PerlHeaderParserHandler` described later. 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.

7.2.6 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>
```

7.2.7 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 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>
```

7.2.8 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);

my %protected = (
    'admin'  => ['gozer'],
    'report' => [qw(gozer 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 *gozer*, */company/report/* can be accessed by users *gozer* 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, 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>
```

7.2.9 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.

7.2.10 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::RequestUtil ();

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.

7.2.11 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;
    use warnings;
    use strict 'refs';
    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!

7.2.12 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 $log_path = catfile Apache::ServerUtil::server_root,
      "logs", "$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/gozer/test.pl
  http://localhost/users/eric/test.pl
  http://localhost/users/gozer/date.pl
```

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

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

7.2.13 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.

8 mod_perl 1.0 to mod_perl 2.0 Migration

8.1 Description

This chapter explains how to migrate from mod_perl 1.0 to mod_perl 2.0.

8.2 Migrating from mod_perl 1.0 to mod_perl 2.0

The following sections discuss what should be done in order to migrate services from mod_perl 1.0 to 2.0 and if possible making the new services based on mod_perl 2.0 back compatible with mod_perl 1.0.

Several configuration directives were renamed or removed. Several APIs have changed, renamed, removed, or moved to new packages. Certain functions while staying exactly the same as in mod_perl 1.0, now reside in different packages. Before using them you need to find out and load those new packages containing them.

Since as of this writing mod_perl 2.0 wasn't released yet, it's possible that certain things have changed after this tutorial has been published. If something doesn't work as explained here, please refer to the documents in the mod_perl distribution or the online version at *http://perl.apache.org/docs/2.0/* for the updated documentation.

8.3 The Shortest Migration Path

mod_perl 2.0 provides two backwards-compatibility layers: one for the configuration files and the other for the code. If you are concerned to preserve the backwards compatibility with mod_perl 1.0, or simply want to try your services under mod_perl 2.0, simply enable the code compatibility layer by adding:

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

at the top of your startup file. The configuration backwards-compatibility is enabled by default.

8.4 Migrating Configuration Files

To migrate the configuration files to the mod_perl 2.0 syntax, you may need to do certain adjustments. Several configuration directives are deprecated in 2.0, but still available for backwards compatibility with mod_perl 1.0. If you don't need the backwards compatibility consider using the directives that have replaced them.

8.4.1 `PerlHandler`

`PerlHandler` was replaced with `PerlResponseHandler`.

8.4.2 `PerlSendHeader`

`PerlSendHeader` was replaced with `PerlOptions +/-ParseHeaders` directive.

```
PerlSendHeader On  => PerlOptions +ParseHeaders
PerlSendHeader Off => PerlOptions -ParseHeaders
```

8.4.3 `PerlSetupEnv`

`PerlSetupEnv` was replaced with `PerlOptions +/-SetupEnv` directive.

```
PerlSetupEnv On  => PerlOptions +SetupEnv
PerlSetupEnv Off => PerlOptions -SetupEnv
```

8.4.4 `PerlTaintCheck`

The taint mode now can be turned on with:

```
PerlSwitches -T
```

It's disabled by default. You cannot disable it once it's enabled.

The default is *Off*. You cannot turn it *Off* once it's turned *On*.

8.4.5 `PerlWarn`

Warnings now can be enabled globally with:

```
PerlSwitches -w
```

8.4.6 `PerlFreshRestart`

`PerlFreshRestart` is a mod_perl 1.0 legacy and doesn't exist in mod_perl 2.0. A full tear-down and startup of interpreters is done on restart.

If you need to use the same *httpd.conf* for 1.0 and 2.0, use:

```
<IfDefine !MODPERL2>
    PerlFreshRestart
</IfDefine>
```

8.5 Code Porting

mod_perl 2.0 is trying hard to be back compatible with mod_perl 1.0. However some things (mostly APIs) have been changed. In order to gain a complete compatibility with 1.0 while running under 2.0, you should load the compatibility module as early as possible:

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

at the server startup. And unless there are forgotten things or bugs, your code should work without any changes under 2.0 series.

However, unless you want to keep the 1.0 compatibility, you should try to remove the compatibility layer and adjust your code to work under 2.0 without it. You want to do it mainly for the performance improvement. The online mod_perl documentation includes a document (*http://perl.apache.org/docs/2.0/user/compat/compat.html*) that explains what APIs have changed and what new APIs should be used instead.

If you have mod_perl 1.0 and 2.0 installed on the same system and the two use the same perl libraries directory (e.g. */usr/lib/perl5*), to use mod_perl 2.0 make sure to load first the `Apache2` module which will perform the necessary adjustments to `@INC`.

```
  use Apache2; # if you have 1.0 and 2.0 installed
  use Apache::compat;
```

So if before loading `Apache2.pm` the `@INC` array consisted of:

```
  /home/gozer/perl/ithread/lib/5.8.0/i686-linux-thread-multi
  /home/gozer/perl/ithread/lib/5.8.0
  /home/gozer/perl/ithread/lib/site_perl/5.8.0/i686-linux-thread-multi
  /home/gozer/perl/ithread/lib/site_perl/5.8.0
  /home/gozer/perl/ithread/lib/site_perl
  .
```

It will now look as:

```
  /home/gozer/perl/ithread/lib/site_perl/5.8.0/i686-linux-thread-multi/Apache2
  /home/gozer/perl/ithread/lib/5.8.0/i686-linux-thread-multi
  /home/gozer/perl/ithread/lib/5.8.0
  /home/gozer/perl/ithread/lib/site_perl/5.8.0/i686-linux-thread-multi
  /home/gozer/perl/ithread/lib/site_perl/5.8.0
  /home/gozer/perl/ithread/lib/site_perl
  .
```

Notice that a new directory was prepended to the search path, so if for example the code attempts to load `Apache::RequestRec` and there are two versions of this module undef */home/gozer/perl/ithread/lib/site_perl/*:

```
          5.8.0/i686-linux-thread-multi/Apache/RequestRec.pm
  5.8.0/i686-linux-thread-multi/Apache2/Apache/RequestRec.pm
```

The mod_perl 2.0 version will be loaded first, because the directory *5.8.0/i686-linux-thread-multi/Apache2* is coming before the directory *5.8.0/i686-linux-thread-multi* in `@INC`.

8.5.1 Finding Which Modules Need To Be Loaded

mod_perl 2.0 splits functionality across many more modules and you have to load these modules before the methods that live in them can be used. So the first step is to figure out which these modules are and `use()` them.

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

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

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

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

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

This prints:

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

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

Since you may need to use this technique quite often you may want to create a handy alias for this technique. For example, C-style shell users can do:

```
   % alias lookup "perl -MApache2 -MModPerl::MethodLookup -e print_method"
```

For Bash-style shell users:

```
   % alias lookup="perl -MApache2 -MModPerl::MethodLookup -e print_method"
```

Once defined the last command line lookup can be accomplished with:

```
  % lookup content_type
```

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

8.6 `ModPerl::Registry` Family

In mod_perl 2.0, `Apache::Registry` and friends (`Apache::PerlRun`, `Apache::RegistryNG`, etc) have migrated into the `ModPerl::` namespace. The new family is based on the idea of `Apache::RegistryNG` from mod_perl 1.0, where you can customize pretty much all the functionality by providing your own hooks. The functionality of the modules `Apache::Registry`, `Apache::RegistryBB` and `Apache::PerlRun` hasn't changed from the user's perspective. All these modules are derived from the `ModPerl::RegistryCooker` class. So if you want to change the functionality of any of the existing sub-classes, or want to "cook" your own registry module it can be done easily. Refer to the `ModPerl::RegistryCooker` manpage for more information.

Here is a typical registry section configuration in mod_perl 2.0:

```
Alias /perl/ /home/httpd/perl/
<Location /perl>
    SetHandler perl-script
    PerlResponseHandler ModPerl::Registry
    Options +ExecCGI
    PerlOptions +ParseHeaders
</Location>
```

As we have explained earlier, the `ParseHeaders` option is needed if the headers are being sent via print() (i.e. without using mod_perl API) and comes as a replacement for the `PerlSendHeader` option in mod_perl 1.0.

Here is a simple registry script that prints the environment variables.

```
file:print_env.pl
-----------------
print "Content-type: text/plain\n\n";
for (sort keys %ENV){
    print "$_ => $ENV{$_}\n";
}
```

Save the file in */home/httpd/perl/print_env.pl* and make it executable:

```
panic% chmod 0700 /home/gozer/modperl/mod_perl_rules1.pl
```

Now issue a request to *http://localhost/perl/print_env.pl* and you should see all the environment variables printed out.

The only change for registry scripts from mod_perl 1.0 is that Perl doesn't `chdir()`'s into the script's directory before executing it. This is because `chdir()` is not a thread-safe function, and as you've learned by now, mod_perl 2.0 should run properly in the threaded environment. To accommodate for this change, the directory of the script is being pushed as the first element in `@INC` for the duration of the script's execution, so relative to the script's directory `require()` calls will succeed. This however doesn't solve the problem for other operations like file `open()` calls, when a relative to the script's directory path is used. In these cases the code needs to be changed to figure out the full path to the file at run time.

8.7 Method Handlers

In mod_perl 1.0 the method handlers could be specified by using the `($$)` prototype:

```
package Bird;
@ISA = qw(Eagle);

sub handler ($$) {
    my($class, $r) = @_;
    ...;
}
```

Starting from Perl version 5.6, you can use subroutine attributes, and that's what mod_perl 2.0 does instead of conventional prototypes:

```
package Bird;
@ISA = qw(Eagle);

sub handler : method {
    my($class, $r) = @_;
    ...;
}
```

See the *attributes* manpage.

mod_perl 2.0 doesn't support the `($$)` prototypes, mainly because several callbacks in 2.0 have more arguments than `$r`, so the `($$)` prototype doesn't make sense anymore. Therefore if you want your code to work with both mod_perl generations, you should use the subroutine attributes.

8.8 `Apache::StatINC` Replacement

`Apache::StatINC` has been replaced by `Apache::Reload`, which works for both mod_perl generations. To migrate to `Apache::Reload` simply replace:

```
PerlInitHandler Apache::StatINC
```

with:

```
PerlInitHandler Apache::Reload
```

However `Apache::Reload` provides an extra functionality, covered in the module's manpage.

9 That’s all folks!

9.1 Thanks

Thanks to TicketMaster for sponsoring some of my work on mod_perl

9.2 References

mod_perl 2.0 information can be found at:

```
http://perl.apache.org/docs/2.0/
```

Further Questions?

Ask at modperl@perl.apache.org

9.3 A Shameless Plug

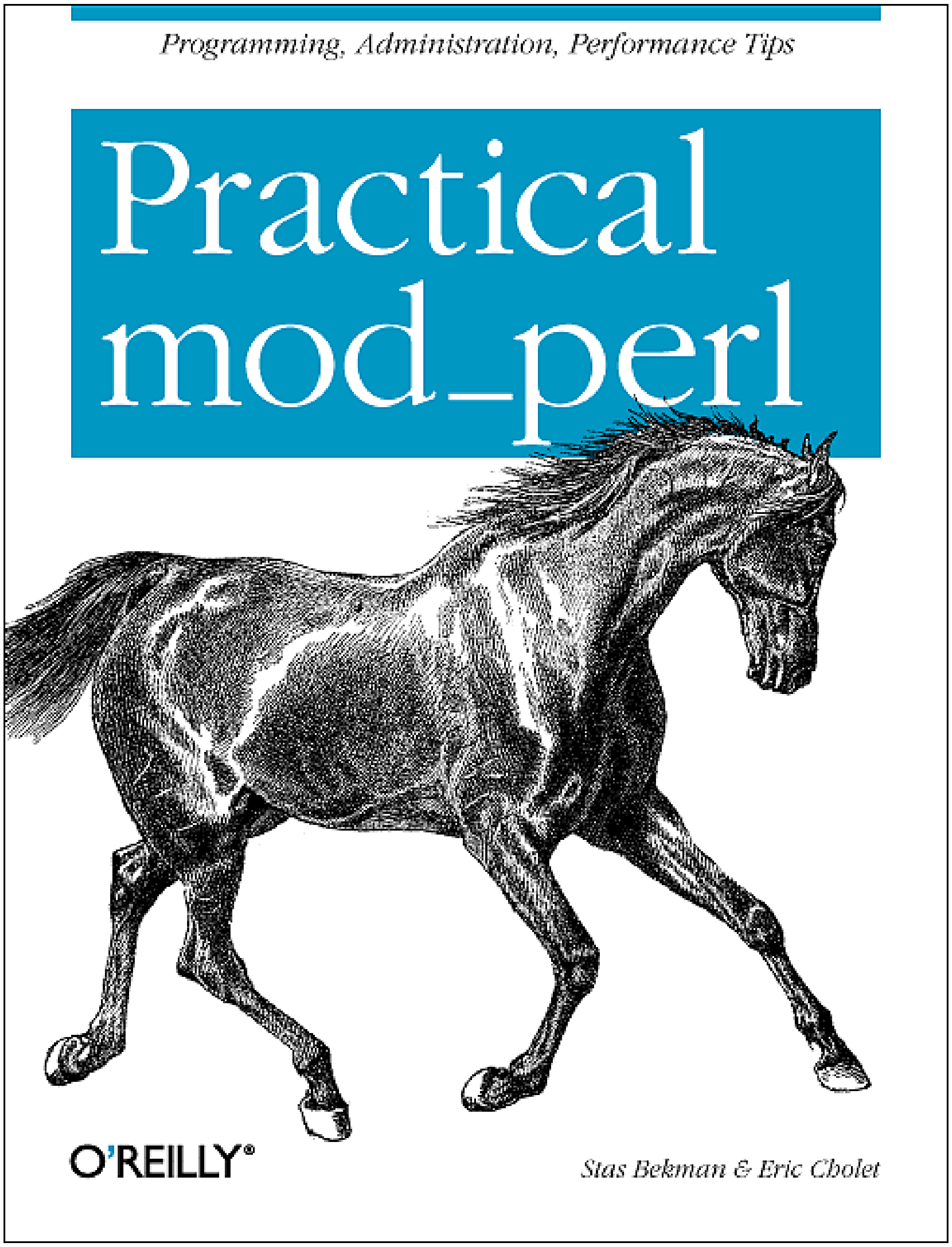

Programming, Administration, Performance Tips
Practical mod_perl
O'REILLY®
Stas Bekman & Eric Cholet

Table of Contents: