Tutorial Slides: 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 talk is available from:
http://gozer.ectoplasm.org/talks/

Last modified Sat Nov 13 09:20:48 2004 GMT

1 Getting Your Feet Wet With mod_perl 2.0

1.1 About

- Prerequisites
- Installation
- Configuration
- Server Launch and Shutdown
- Registry Scripts
- Handler Modules

1.2 Prerequisites

- Apache 2.0 is required.
- Perl, depending on MPM:
  - prefork: 5.6.0, better 5.6.1
  - threaded: 5.8.0 + ithreads
- The installation details are in the handouts

1.3 mod_perl Installation

```
% lwp-download \
  http://perl.apache.org/dist/mod_perl-2.0.01.tar.gz
% 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
% make && make test && make install
```

- MP_APXS is a full path to the apxs executable

1.4 Configuration

- Enable DSO:

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

- Find 2.0 modules

```
PerlModule Apache2
```

- Enable the 1.0 compatibility layer

```
PerlModule Apache::compat
```

1.5 Server Launch and Shutdown

- Start:

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

% tail -f $HOME/httpd/prefork/logs/error_log

[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
```

- Stop:

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

1.6 Registry Scripts

- Configuration

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

- Restart the server

- A simple script:

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

- Make it executable and readable:

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

- Request:

```
% lwp-request http://localhost/perl/rock.pl
mod_perl 2.0 rocks!
```

1.7 Handler Modules

```
# /home/httpd/httpd-2.0/perl/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;
```

- Adjust @INC via a startup file

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

- loaded from *httpd.conf*:

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

- Configure:

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

- Restart server

- Issue request:

```
% lwp-request http://localhost/rocks
mod_perl 2.0 rocks!
```

2 New Concepts

2.1 About

- Exceptions
- Bucket Brigades

2.2 Exceptions

- Apache and APR API return a status code for almost all methods
- In C you must check return value of each call
- It makes it really hard/slow to prototype things
- It makes the code hard to read
- It’s easy to forget to check the return values -- causing problems later
- It complicates the API, when you may want to return other values

- `APR::Error` handles APR/Apache/mod_perl exceptions for you, while leaving you in control.
- Instead of returning status codes, we throw exceptions

- If you don’t catch those exceptions, everything works transparently

```
my $rlen = $sock->recv(my $buff, 1024);
```

- Perl will intercept the exception object and `die()` with a proper error message.

```
[Wed Jun 09 07:25:25 2004] [error] [client 127.0.0.1]
APR::Socket::recv: (11) Resource temporarily unavailable
at .../TestError/runtime.pm line 124
```

- If you want to catch an exception -- you can.

```
use APR::Const -compile => qw(TIMEUP);                                  #
my $tries = 0;
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;
    }
}
warn "read $rlen bytes\n";
```

- `APR::Error` uses Perl operator overloading:
  - in boolean and numerical contexts -- gives the status code
  - in the string context -- the full error message
- it is called `APR::Error` as it is used by
  - mod_perl
  - APR apps written in Perl

2.3 Bucket Brigades

- Apache 2.0 implements request and response data flow filtering
- Modules can filter each other's output
- No need to modify Apache to accomodate SSL, compressions, transformation filters.
- The *Bucket Brigades* technology was introduced to make IO filtering efficient and avoid unnecessary copying.

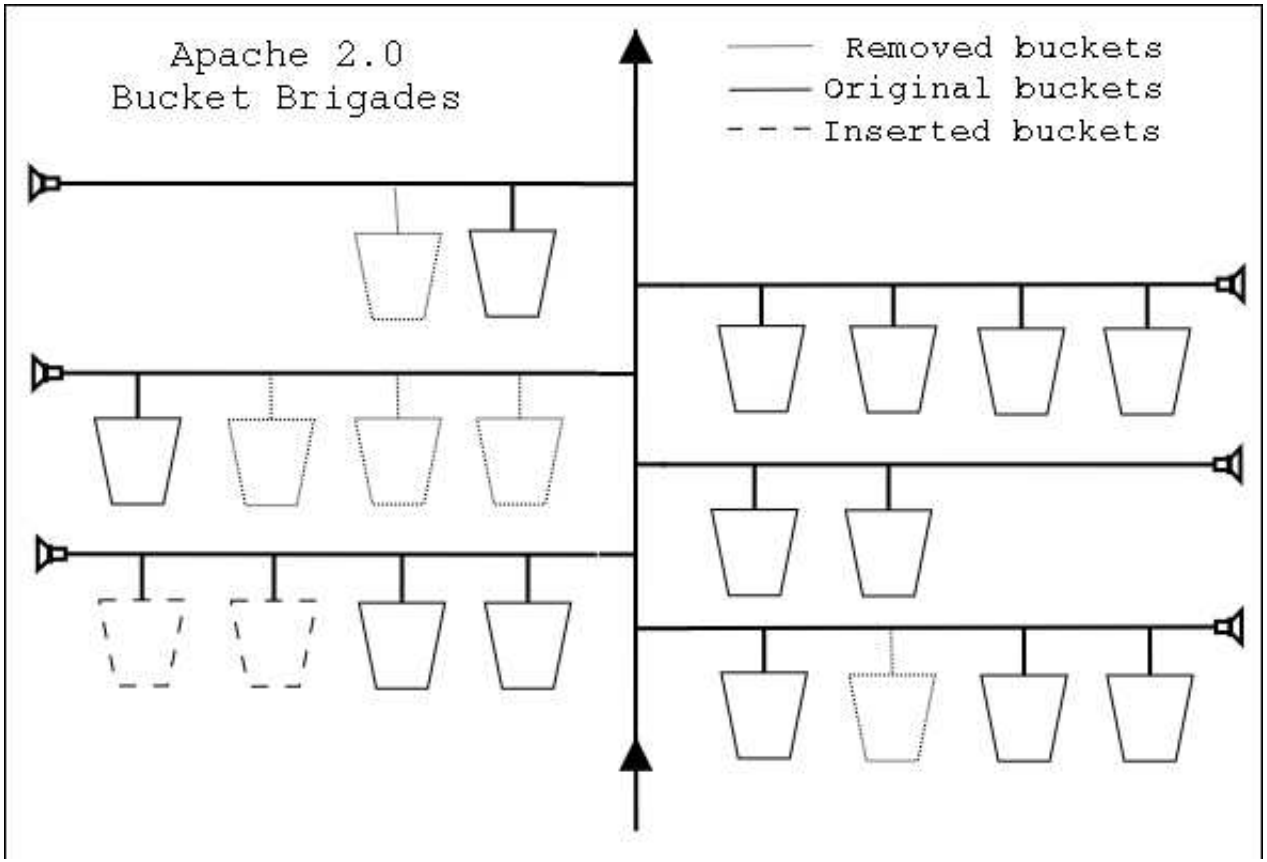
Buckets:

- 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.
- Bucket types: files, data blocks, flush and end of stream indicators, pools, etc.
- To manipulate a bucket one doesn't need to know its internal representation.

Bucket Brigades:

- A stream of data is represented by bucket brigades.
- Filters manipulate brigades one by one (adding/modifying/removing buckets)
- ... and pass the brigade to the next filter on the stack

- Here’s an imaginary bucket brigade after it has passed through several filters.
- Some buckets were removed, some modified and some added.



- More about BBs when we talk about protocols and filters

3 Introducing mod_perl Handlers

3.1 About

- Handler Anatomy
- mod_perl Handlers Categories
- Single Phase’s Multiple Handlers Behavior

3.2 Handler Anatomy

- Apache distinguishes between numerous phases
- Each phase provides a hook: *ap_hook_<phase_name>*
- These hooks are used by modules to alter the default Apache behavior
- Hooks are usually referred to as handlers or callbacks
- Naming convention: `PerlFooHandler`
- e.g. `PerlResponseHandler` configures the response callback.

```
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;
```

- Configuring the response handler:

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

- A request to *http://localhost/time* returns the curent time

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

- 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	

The types:

- VOID
  - Executed in the order they have been registered disregarding their return values.
  - Though in mod_perl they are expected to return `Apache::OK`.
- RUN_ALL
  - Executed in the order they have been registered until the first handler that returns something other than `Apache::OK` or `Apache::DECLINED`.

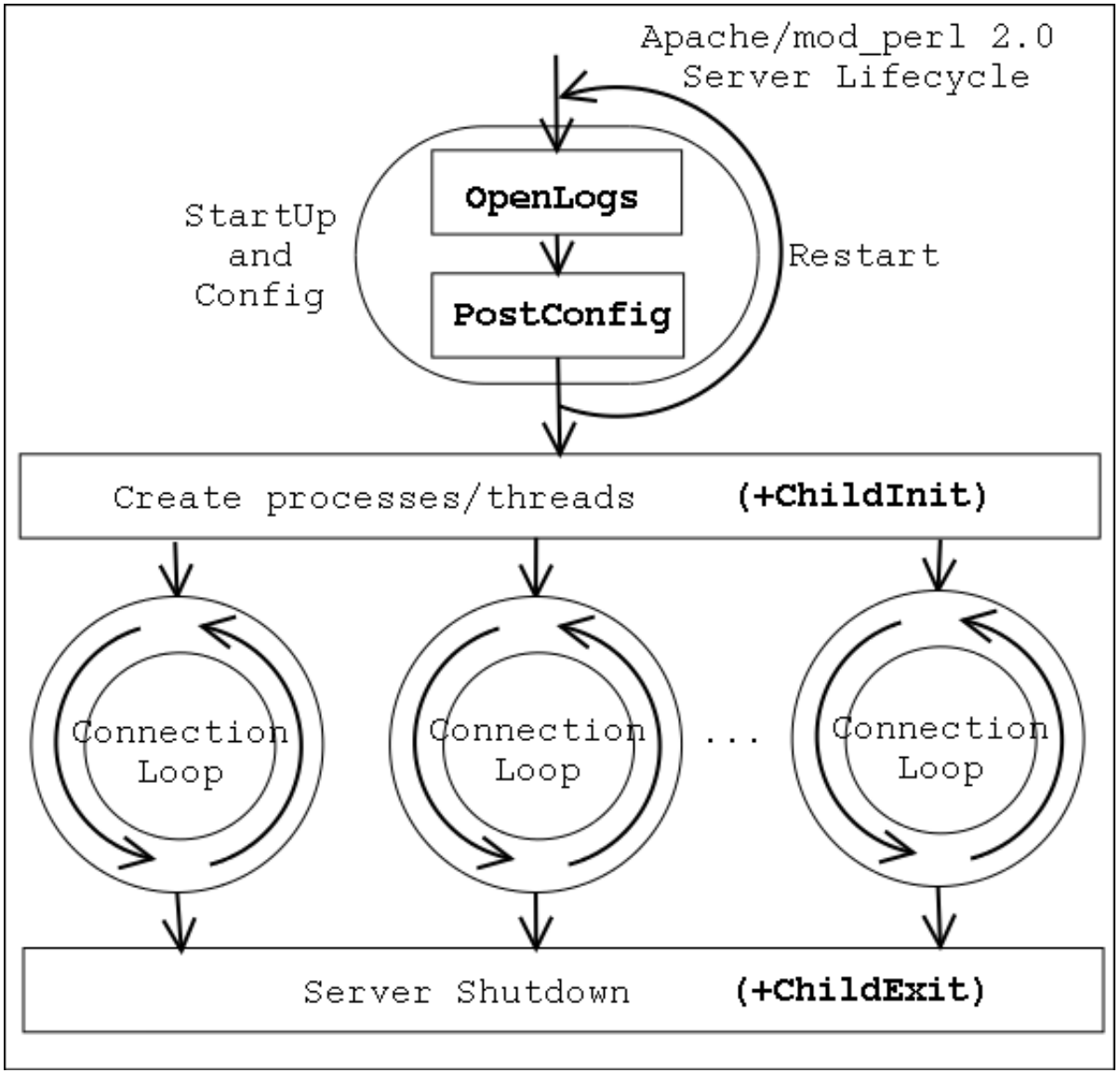
- RUN_FIRST
  - Executed in the order they have been registered until the first handler that returns something other than `Apache::DECLINED`.
  - If the return value is `Apache::DECLINED`, the next handler in the chain will be run.
  - If the return value is `Apache::OK` the next phase will start.
 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

4 Server Life Cycle Handlers

4.1 About

- Server Life Cycle
- Startup Phases Demonstration Module
- PerlOpenLogsHandler
- PerlPostConfigHandler
- PerlChildInitHandler
- PerlChildExitHandler

4.2 2.0 Server Life Cycle



1. Parse the configuration file, open_logs, post_config
2. Restart to test graceful restarts
3. Parse the configuration file, open_logs, post_config
4. Spawn workers: procs, threads, mix
    - Run child_init for each spawned process
5. Process requests over connections
6. Shutdown: child_exit

4.2.1 Startup Phases Demonstration Module

```
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:

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

% bin/apachectl start

```
# 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
```

- Apache restarts itself

```
% bin/apachectl stop

[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
--------------------
```

- All processes run the END block on shutdown

- The presented behavior varies from MPM to MPM.
- I used worker MPM for this demo
- MPMs, like winnt, may run *open_logs* and *post_config* more than once
- the END blocks may be run more times, when threads are involved.
- The only sure thing -- each of these phases run at least once

5 Protocol Handlers

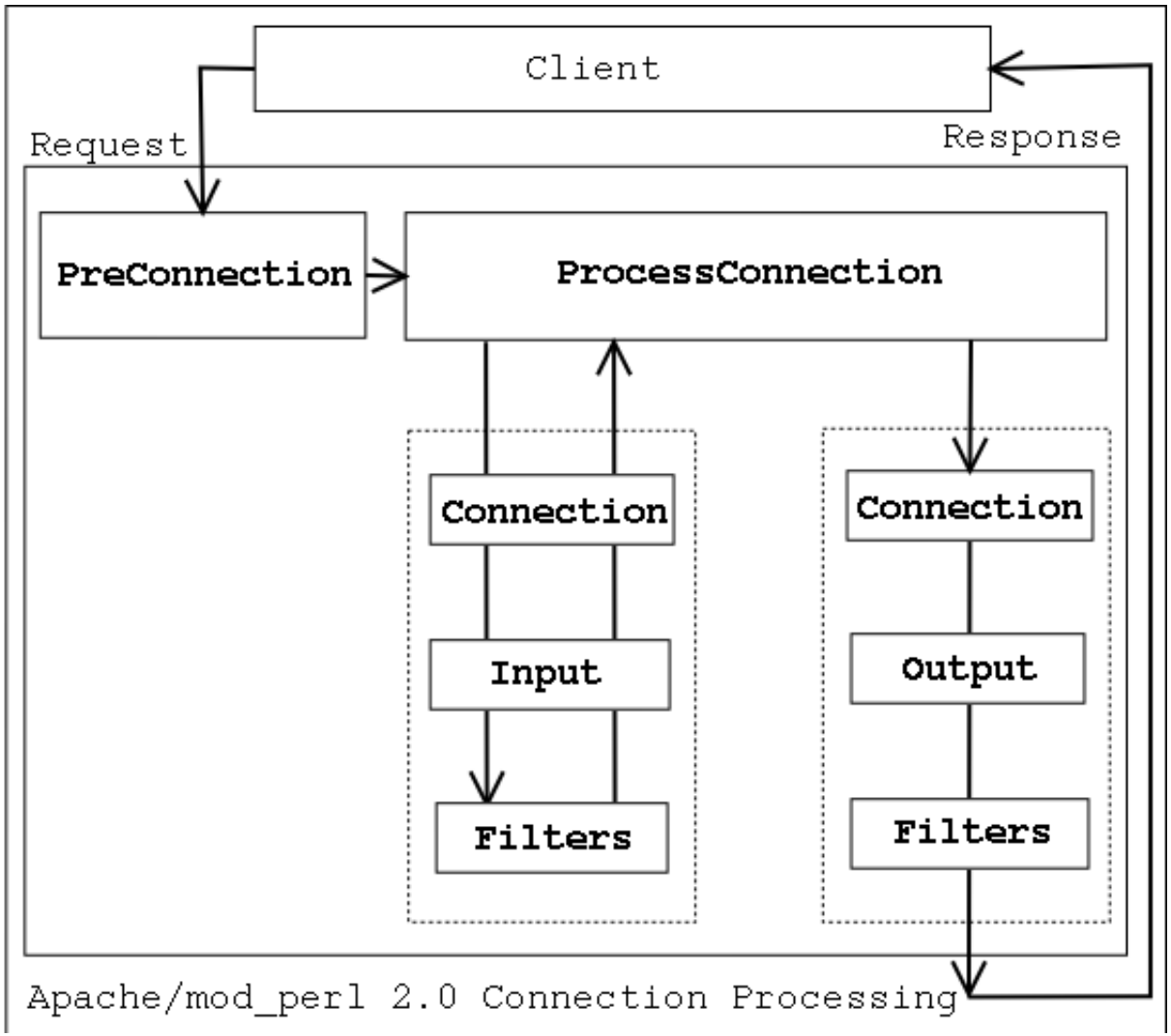
5.1 About

- Connection Cycle Phases
- PerlPreConnectionHandler
- PerlProcessConnectionHandler
- Socket-based Protocol Module
- Bucket Brigades-based Protocol Module

5.2 Connection Cycle Phases

- Each child server 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 a response includes several images.

Connection Life Cycle diagram:



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.

Apache::Reload:

- In mod_perl 1.0 `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 general phase
- So now we can invoke the `Apache::Reload` handler during the *pre_connection* phase if the interpreter's scope is:

```
PerlInterpScope connection
```

- Though this is not good for a production server,
- where 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.

Skeleton:

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

- A *pre_connection* handler accepts two arguments:
    - connection record
    - and socket objects

Block by IP example:

```
package MyApache::BlockIP2;                                    #

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;
```

- Configuration

```
PerlPreConnectionHandler MyApache::BlockIP2
```

- Apache simply drops the connection if the IP is blacklisted
- Almost no resources are wasted

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.

process_connection skeleton:

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

- arg1: a connection record object
- a socket object: retrieved from $c
- must set the socket to blocking IO mode

- Let’s look at the following two examples of connection handlers:

  1. Using the connection socket to read and write the data.

  2. Using bucket brigades to accomplish the same and allow for connection filters to do their work.

5.2.2.1 Socket-based Protocol Module

- The `MyApache::EchoSocket` module simply echoes the data read back to the client.
- This module's implementation works directly with the connection socket and therefore bypasses connection filters if any.

Configuration:

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

- use the `Listen` and `<VirtualHost>` directives to bind to the non-standard port **8010**:

Demo:

```
panic% httpd
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.
```

The code:

```
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;

    $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;
```

5.2.2.2 Bucket Brigades-based Protocol Module

- 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.

Configuration:

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

Demo:

```
panic% httpd
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
```

- As you can see the response now was all in lower case, because of the output filter.

The code:

```
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;

    my $last = 0;
    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->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);
            last if $last;
        }

        $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;
```

- A simplified pseudo-code handler:

```
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);
}
```

Use `fflush` -- replace:

```
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);                      #
```

- This handler could be much simpler, since we don't modify the data.

```
$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;
```

- but it won't work in the telnet mode, since `/[\r\n]/` that we type to end the line will keep this loop running forever

- And here is another version which slurps all the data and works with telnet

```
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 $data =~ /^[\r\n]+$/;
    $bb->insert_tail(APR::Bucket->new($ba, $data));

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

- We will discuss the filters next

6 Input and Output Filters

6.1 About

- Your First Filter
- I/O Filtering Concepts
  - Two Methods for Manipulating Data
  - HTTP Request Versus Connection Filters
  - Multiple Invocations of Filter Handlers
  - Blocking Calls

- mod_perl Filters Declaration and Configuration
  - Filter Priority Types
  - `PerlInputFilterHandler`
  - `PerlOutputFilterHandler`
  - `PerlSetInputFilter`
  - `PerlSetOutputFilter`
  - HTTP Request vs. Connection Filters
  - Filter Initialization Phase

- All-in-One Filter
- Input Filters Examples
- Output Filters Examples

6.2 Your First Filter

- You certainly already know how filters work.
- That’s because you encounter filters so often in real life.


Carrera

Why using I/O filters?

- Instead of making the response handler code complex
- adjust its output with filters:
  - each doing a simple transformation
- and can be:
  - stacked
  - repeated
  - added/removed dynamically at run-time

A simple obfuscation filter:

- turn HTML pages into a one-liner
- disregarding cases where new lines must be preserved,
  - as in `<pre>...</pre>`
- config:

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

```
package MyApache::FilterObfuscate;                                    #

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

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

sub handler {
    my $f = shift;

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

    while ($f->read(my $buffer, 1024)) {
        $buffer =~ s/[\r\n]//g;
        $f->print($buffer);
    }
    return Apache::OK;
}
1;
```

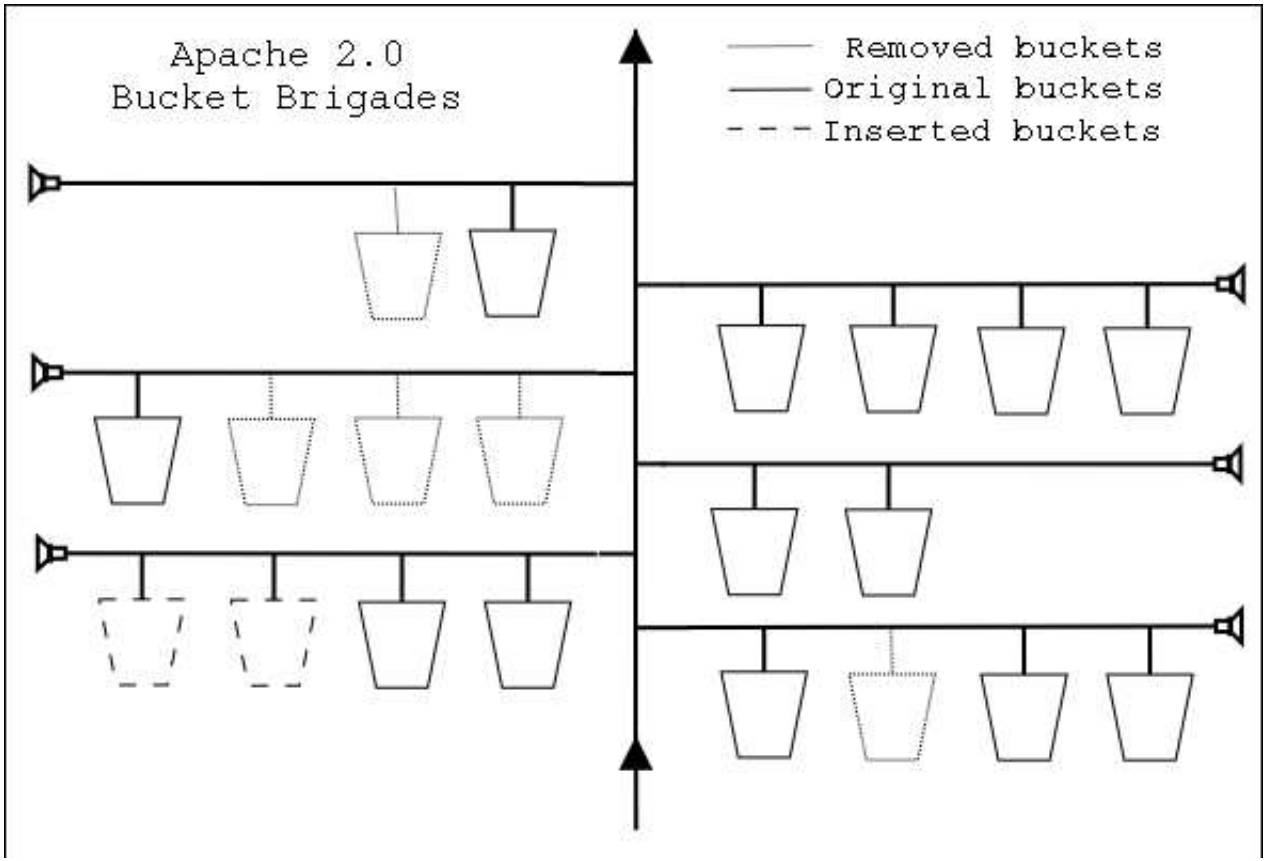
- Will talk about details shortly...

6.3 I/O Filtering Concepts

- Before getting to the real filter examples
- a few concepts to help you wrap your head around ...

6.3.1 2 Ways to Manipulate 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*.
- Both input and output filters, filter the data in bucket brigades.


Apache 2.0
Bucket Brigades
Removed buckets
Original buckets
Inserted buckets

mod_perl 2.0 filters can:

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

6.3.2 HTTP Request vs Connection Filters

HTTP Request filters

- are applied on:
  - HTTP requests body (POST) (if consumed by the content handler)
  - HTTP response body
- HTTP headers are not passed through the HTTP Request filters.

Connections filters:

- are applied at the connection level.
- can modify all the incoming and outgoing data.
    - including HTTP headers if HTTP protocol is used

6.3.3 Multiple Invocations of Filter Handlers

- a filter gets invoked as many times as the number of bucket brigades sent from an upstream filter or a content provider.
- Response handler example, buffered STDOUT ($|==0):

```
$r->print("foo");
$r->rflush;
$r->print("bar");
```

- Apache generates 3 bucket brigades => 3 filter invocations

- print and rflush => two buckets:

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

- print => one bucket:

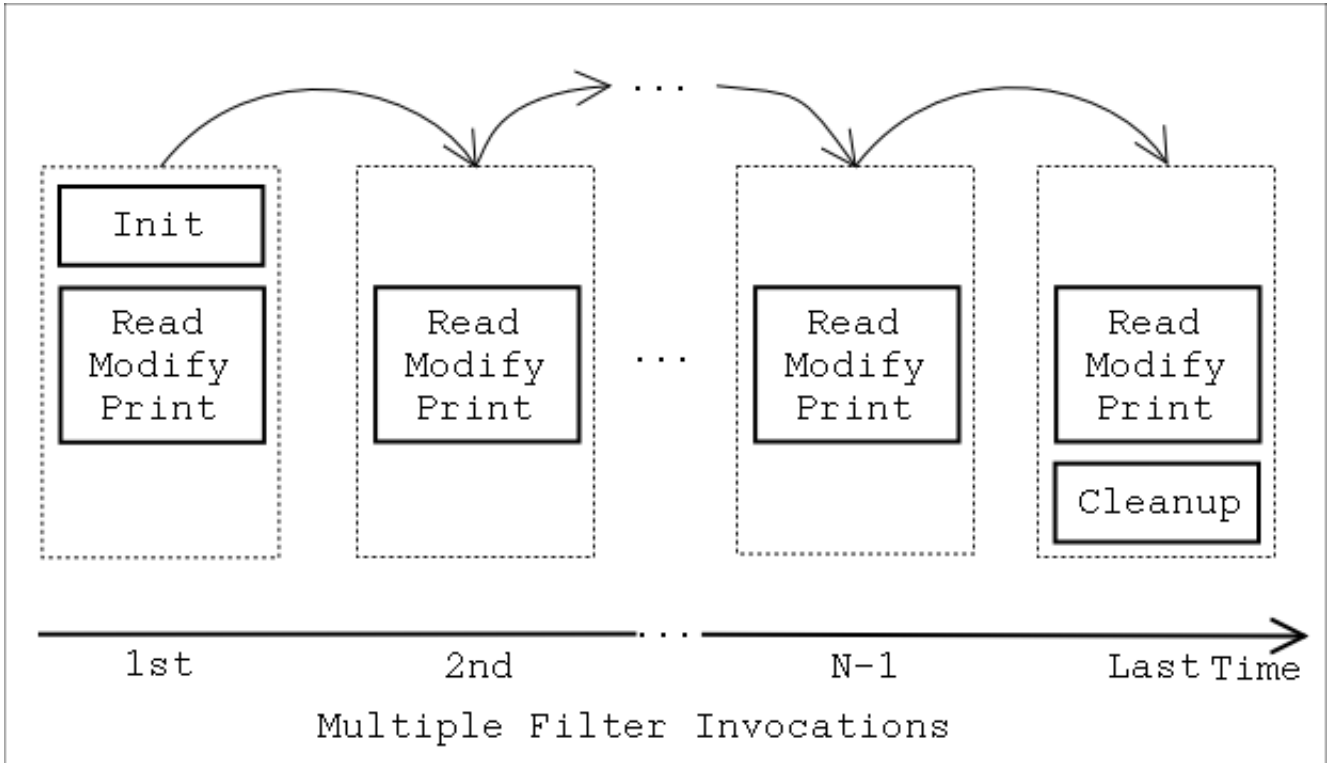
```
bucket type        data
---------------------
1st    transient   bar
```

- when the response handler returns => one bucket:

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

- EOS may come as a last bucket in bb with data

Three Filter Stages:



Filter skeleton demonstrating all 3 stages:

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

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

    process($f);        # runs on all invocations

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

    return Apache::OK;
}
```

Init example:

- response filters changing data length have to unset the C-L header (runs once)

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

Processing example:

- lower the data case (runs on all invocations)

```
while ($f->read(my $data, 1024)) {
    $f->print(lc $data);
}
```

- in complex cases may need to store data in context between filter invocations

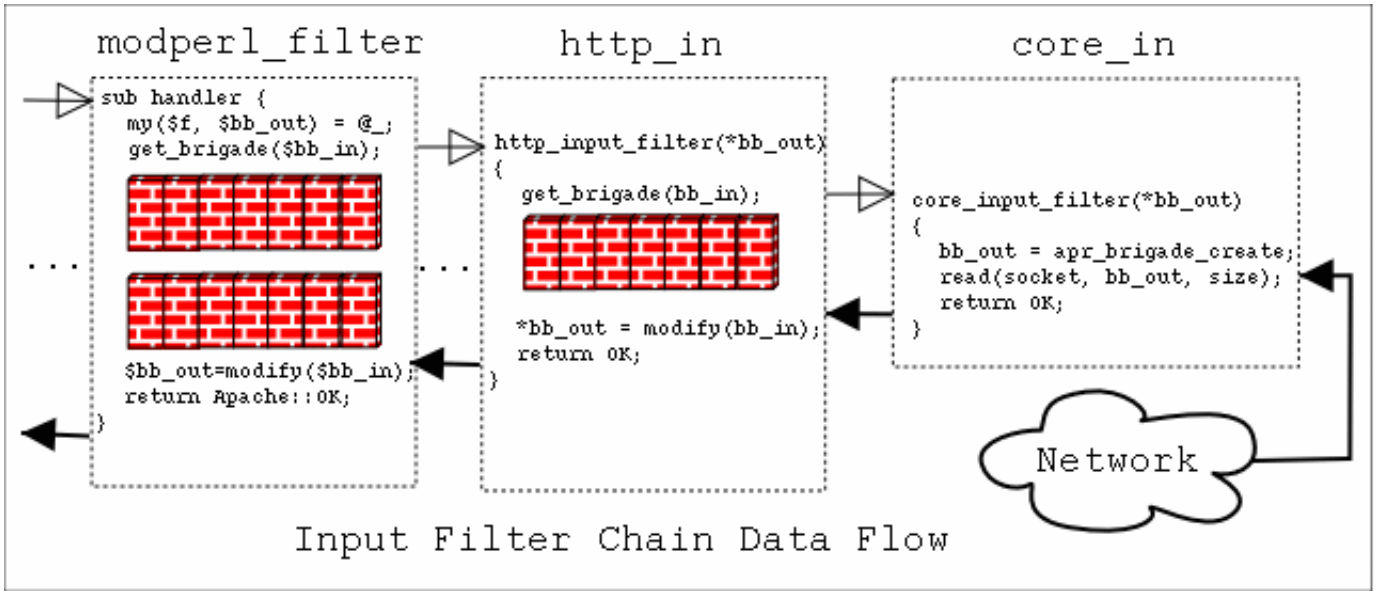
Finalization example:

- After all data was sent out, print how many times the filter was invoked

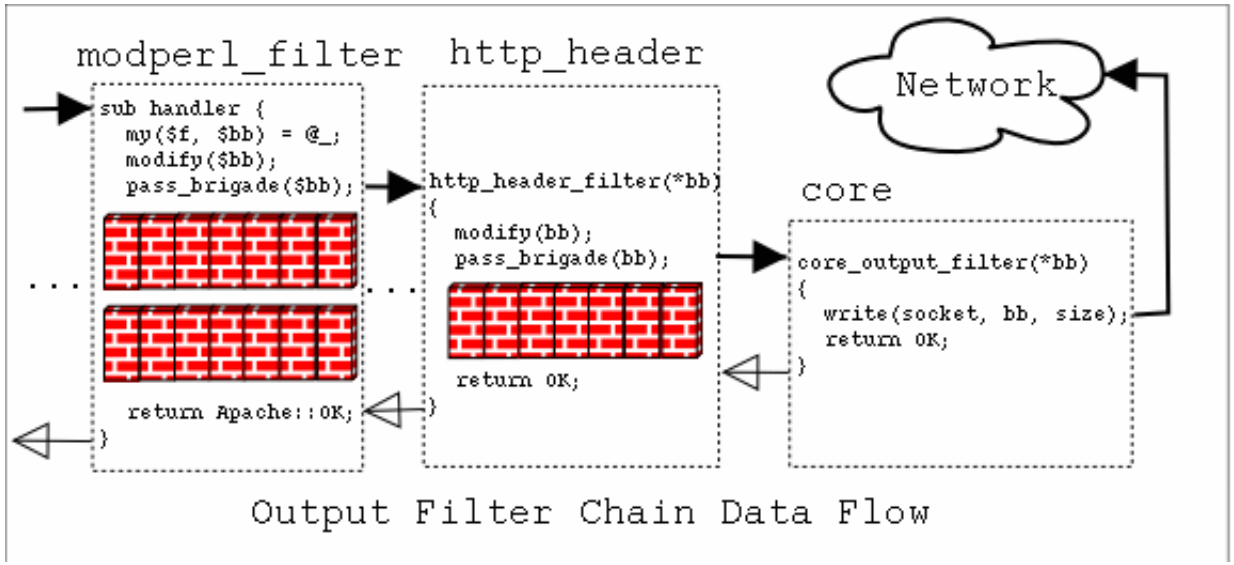
```
my $ctx = $f->ctx;
$ctx->{invoked}++;
$f->ctx($ctx); # store

while ($f->read(my $data, READ_SIZE)) {
    $f->print(lc $data);
}

if ($f->seen_eos) {
   $f->print("Filter invocated $ctx->{invoked} times");
}
```

6.3.4 Blocking Calls

- Data propogates in stages through the filter chain (there is no pipeline)
- The input filter's `get_brigade()` call blocks



- The output filter's `pass_brigade()` call blocks



Transparent input filter:

```
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;
}
```

Transparent output filter:

```
sub out {
    my ($f, $bb) = @_;

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

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

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

- Filters are inserted into the filter chain based on their priority
- mod_perl provides 2 filters types:

Handler	Priority	Value
FilterRequestHandler	AP_FTYPE_RESOURCE	10
FilterConnectionHandler	AP_FTYPE_PROTOCOL	30

- Apache provides others as well

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`

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`

6.4.4 PerlSetInputFilter

- mod_perl equivalent for `SetInputFilter`
- makes sure that mod_perl and non-mod_perl input filters of the same priority get inserted in the order they are configured

- For example:

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

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

6.4.5 PerlSetOutputFilter

- mod_perl equivalent for `SetOutputFilter`
- makes sure that mod_perl and non-mod_perl output filters of the same priority get inserted in the order they are configured

- For example:

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

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

- If filters have different priorities the insertion order might be different:

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

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

6.4.6 HTTP Request vs. Connection Filters

- method attributes set the filter type:

```
sub handler : FilterRequestHandler    { ... }
sub handler : FilterConnectionHandler { ... }
```

- HTTP Request filter handlers are declared using the `FilterRequestHandler` attribute.

- HTTP 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 sub-class `Apache::Filter`, others only need to:

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

- 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>
```

- Connection filter handlers are declared using the `FilterConnectionHandler` attribute.

- Connection input and output filters skeleton:

```
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>
```

- 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 mapping to buckets and bucket brigades
  - and a simpler, stream-oriented interface
- Following examples will explain the difference

6.4.7 Filter Initialization Phase

- There is a special attribute duet `FilterHasInitHandler`/`FilterInitHandler` but it's left for your offline reading.

6.5 All-in-One Filter

- The `MyApache::FilterSnoop` handler silently snoops on the data that goes through request and connection filters, in the input and output modes.

- First let's develop a simple response handler that simply dumps the request's *args* and *content* as strings:

```
package MyApache::Dump;

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

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

```
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;
}
```

```
 sub content {

    # see your handouts, the details are unimportant here

 }

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:

```
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;
```

- We want to use this somewhat complicated filter to visualize how various kind of filters work.
- Since this module combines several kind of filters it’s the best to try to understand its implementation after you understand standalone filters’ implementation.
- At this stage what’s important is to see it at work.

- 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>
```

- If we issue the following request:

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

- the response doesn't get affected by the snooping filter
- though we can see all the diagnostics in *error_log*

- 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
[]
```

- The connection input filter is run before the request input filter.

- Both filters see the same data, since they don’t modify it
- The bucket of type *EOS* indicates the end of stream.

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

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

- Next, Apache injects the HTTP headers:

```
>>> 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

]
```

- the first response body's brigade inside the connection output filter:

```
>>> 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
[
]
```

- Finally the output is flushed,
- to make sure that any buffered output is sent to the client:

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

]
    o bucket 2: EOS
[]
```

- 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.
- the HTTP request input filter is called only if there is some POSTed data to read
- If you run the same request without POSTing any data or simply running a GET request, the request input filter won’t be called.

6.6 Input Filters

- mod_perl supports 2 kinds of input filters:

    1. Connection input filters

    2. HTTP Request input filters

6.6.1 Connection Input Filters

- Let's write a poor man's s/GET/HEAD/ rewrite handler
- The handler looks for the data like:

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

- and turns it into:

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

```
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;
```

- For example, consider the following response handler:

```
package MyApache::RequestType;                                    #

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_:8005>
    <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:8005/"); \
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.

- 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 similar to connection filters, but have an access to a request object and they don’t see the headers.

6.6.3 Bucket Brigade-based Input Filters

- Here is the request input filter that lowercases the request's body:

```
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;
```

- the MyApache::Dump response handler dumps the query string and the content body as a response

- Using the following configuration:

```
<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.

- The query string didn't change, since the filter didn't operate on headers.

6.6.4 Stream-oriented Input Filters

```
package MyApache::InputRequestFilterLC2;                    #

use base qw(Apache::Filter);

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

sub handler : FilterRequestHandler {
    my $f = shift;

    while ($f->read(my $buffer, 1024)) {
        $f->print(lc $buffer);
    }

    Apache::OK;
}
1;
```

- Configuration:

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

- We get the same results as from using the bucket brigades filter

6.7 Output Filters

- mod_perl supports 2 kinds of output filters:
    1. Connection input filters
    2. HTTP Request input filters

6.7.1 Connection Output Filters

- Connection filters are similar to HTTP Request filters,
- but can see **all** the data that is going through the server.
- We will concentrate on HTTP request filters

6.7.2 HTTP Request Output Filters

- Output filters can be written using the bucket brigades manipulation or the simplified stream-oriented interface.
- Here is a response handler that send two lines of output in a single string:

  1. numerals: 1234567890
  2. the English alphabet

```
package MyApache::SendAlphaNum;                                    #

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 request output filter is to reverse every line of the response, preserving the new line characters in their places.

- Since we want to reverse characters only in the response body we will use the HTTP request output filter.

6.7.3 Stream-oriented Output Filter

```
package MyApache::FilterReverse1;                                    #

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

sub handler : FilterRequestHandler {
    my $f = shift;

    while ($f->read(my $buffer, 1024)) {
        for (split "\n", $buffer) {
            $f->print(scalar reverse $_);
            $f->print("\n");
        }
    }
    Apache::OK;
}
1;
```

- 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

6.7.4 Bucket Brigade-based Output Filters

- The bucket brigades filtering API implementation:

```
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.

7 HTTP Handlers

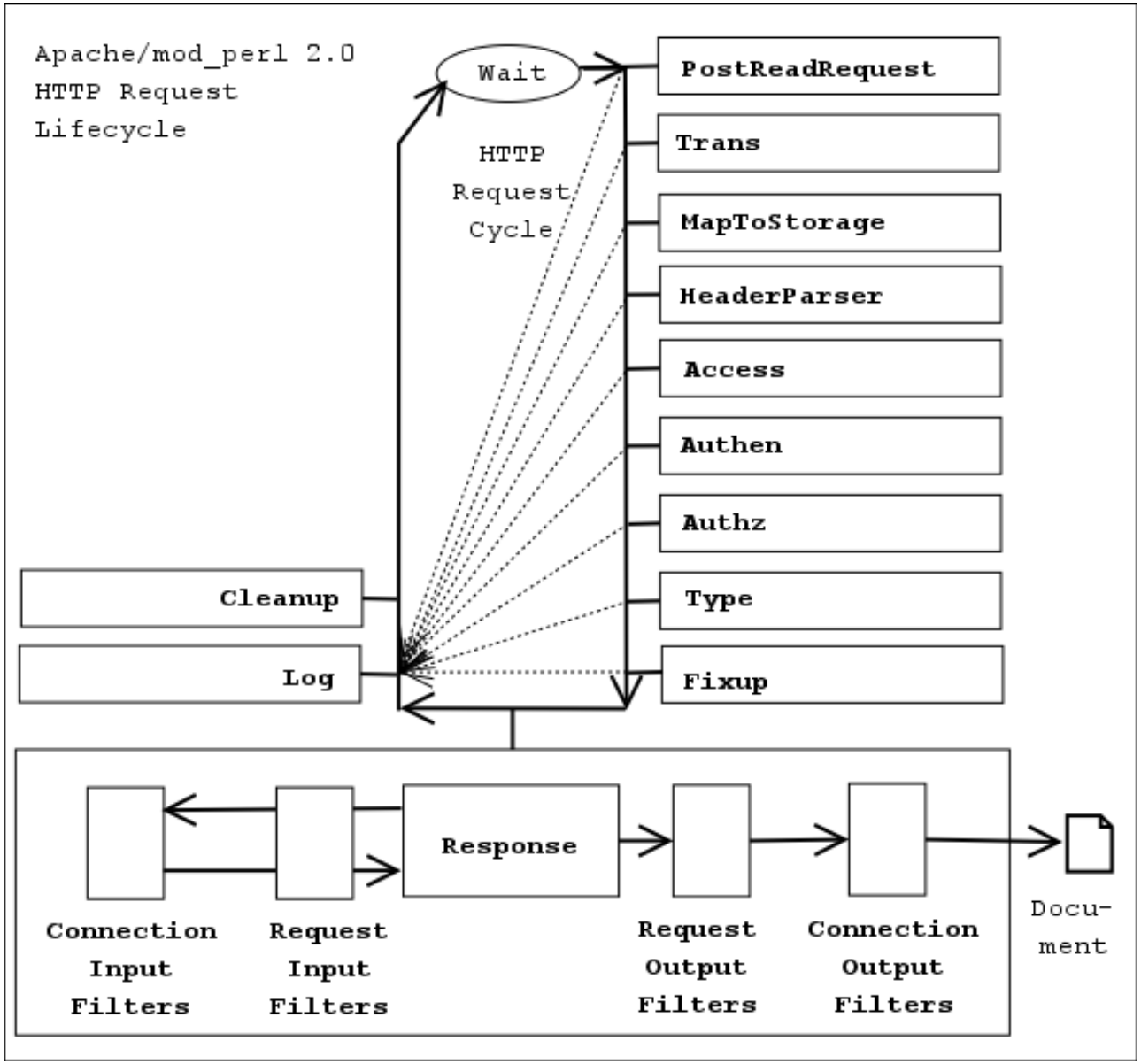
7.1 About

- HTTP Request Cycle Phases
- PerlPostReadRequestHandler
- PerlTransHandler
- PerlMapToStorageHandler
- PerlHeaderParserHandler
- PerlInitHandler
- PerlAccessHandler

- PerlAuthenHandler
- PerlAuthzHandler
- PerlTypeHandler
- PerlFixupHandler
- PerlResponseHandler
- PerlLogHandler
- PerlCleanupHandler

7.2 HTTP Request Cycle Phases

- Almost indentical to mod_perl 1.0. The differences are:
  - new phase: *PerlMapToStorageHandler*
  - rename: `PerlHandler` => `PerlResponseHandler`
  - `PerlResponseHandler` now includes filtering
- Here is the HTTP request life cycle in mod_perl 2.0:


Apache/mod_perl 2.0
HTTP Request
Lifecycle
Wait
HTTP
Request
Cycle
PostReadRequest
Trans
MapToStorage
HeaderParser
Access
Authen
Authz
Type
Fixup
Cleanup
Log
Response
Connection
Input
Filters
Request
Input
Filters
Request
Output
Filters
Connection
Output
Filters
Docu-
ment

- an HTTP request is processes by 12 phases:
  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

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

- The following `ModPerl::Registry` script prints when the last time *httpd.conf* has been modified, compared to the start of the request process time:

```
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);
```

- The script reports incorrect time most of the time
- Because `-M` reports the difference between file's modification time and `$^T`
- Under mod_perl `$^T` is set when the process starts and doesn't change after that.
- Solution: Reset `$^T` on each request

```
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 `$r->request_time` 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 manipulate 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.
- Also 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` / config scope is SRV.

mod_rewrite ala mod_perl:

- e.g. if the previously static pages are now autogenerated, and

```
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
```

- we can keep using the old URI, transparent to *news.pl* and users

```
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;
```

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

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

7.2.3 PerlMapToStorageHandler

- The *map_to_storage* phase is used to translate 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.
- most likely you want to shortcut this phase for mod_perl
- This phase is of type `RUN_FIRST` / config scope is `SRV`.

- The Don’t-waste-time handler:

```
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;
```

- config:

```
PerlMapToStorageHandler MyApache::NoTranslation
```

- same handler but support TRACE calls

```
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:

```
$r->filename(__FILE__);
```

- this can be done in `PerlTransHandler` as well

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.
- e.g., this phase can be used to block evil clients targeting certain resources, while few resources were wasted so far.
- This phase is of type `RUN_ALL`.
- The handler's configuration scope is `DIR`.

- Apache handles the `HEAD`, `GET`, `POST` and several other HTTP methods.
- But you can invent your own methods and make Apache accept them.
- e.g., emails are very similar to HTTP messages: they have a set of headers and a body, sometimes a multi-part body.
- `MyApache::SendEmail` extends HTTP by adding a support for the `EMAIL` method.
- It enables the new extension and pushes the real content handler during the `PerlHeaderParserHandler` phase:

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

```
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);
```

```
    $msg->send;
}

sub content {                                                    #
    my $r = shift;

    # unimportant here, see your handouts

    return $buf;
}
1;
```

- 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:

```
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";
```

7.2.5 PerlInitHandler

- When configured inside any container directive, except `<VirtualHost>`, this handler is an alias for `PerlHeaderParserHandler` described earlier.
- Otherwise it acts as an alias for `PerlPostReadRequestHandler` described earlier.
- It is the first handler to be invoked when serving a request.
- This phase is of type `RUN_ALL`.
- The best example here would be to use `Apache::Reload` which takes the benefit of this directive.

- Usually `Apache::Reload` is configured as:

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

- which during the current HTTP request will monitor and reload all `MyApache::*` modules that have been modified since the last HTTP request.

- However if we move the global configuration into a `<Location>` container:

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

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

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`.

- This handler denies requests made from IPs on the blacklist:

```
package MyApache::BlockByIP;                                    #

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;
```

- 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 example handler verifies that:

```
SECRET_LENGTH == length join " ", $username, $password;
```

```
package MyApache::SecretLengthAuth;                                    #

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;
```

- To enable this handler you have to tell Apache:
  - what authentication scheme to use (`AuthType`: `Basic` or `Digest`)
  - what's the authentication realm (`AuthName`: any string)
  - the `Require` directive is needed to specify which usernames are allowed to authenticate. If you set it to `valid-user` any username will do.

```
<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`.

- The `MyApache::SecretResourceAuthz` handler grants access to certain resources only to certain users who have already properly authenticated:

```
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;
```

- The configuration is similar to `PerlAuthenHandler`, 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>
```

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.
- e.g., `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`.
- When overriding the default *type_checker* handler, which is usually the mod_mime handler, don't forget to set the response handler:

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

- 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`.

- This fixup handler 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.

```
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;
```

- 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:

```
<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`.

- This handler prints itself:

```
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 is always executed.
- Even if the previous phases were aborted.
- At this phase log handlers usually log various information about the request and the response.
- This phase is of type `RUN_ALL`.
- The handler's configuration scope is `DIR`.

- The following handler logs the request data into user-specific files.
- We assume that all URIs include the username in the form of: */users/username/*

```
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;
```

- Configuration:

```
<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’s a mod_perl convenience invention
- It runs immediately after the request has been served
- and before the request object is destroyed.
- Used to run cleanup code
- or to do some time consuming work after the client has gone
- This phase is of type `RUN_ALL` / config scope is `DIR`

Usage:

- a `cleanup()` callback accepts `$r` as its only argument.

```
PerlCleanupHandler MyApache::Cleanup
```

or:

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

- a `cleanup()` callback accepts an optional arbitrary argument

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

Example: delete a temporary file in the cleanup handler

```
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 requesting */cleanup1*
  - the contents of the current directory will be printed
  - and once the request is over the temporary file is deleted.

The problem in Multi-processes/threads env

- `MyApache::Cleanup1` is problematic for it uses the same filename
- Better use a unique filename ($$ + APR::OS::thread_current())
- Even better, use a non-predictable name: rand, `APR::UUID`
- But how do we pass the filename to the cleanup handler?
- `MyApache::Cleanup2` solves the problem with:

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

```
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>
```

8 mod_perl 1.0 to mod_perl 2.0 Migration

8.1 About

- Configuration Directives Changes
- mod_perl API Changes
- mod_perl Modules Changes

8.2 Migrating from mod_perl 1.0 to mod_perl 2.0

- Some configuration directives were renamed or removed
- Several APIs have changed, renamed, removed, or moved to new packages.
- ...
- But there is a back-compatibility layer!

8.3 The Shortest Migration Path from 1.0

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

- Certain Configuration directives and APIs have changed
- See http://perl.apache.org/docs/2.0/user/compat/compat.html

8.4 Migrating Configuration Files

- Several configuration directives are deprecated in 2.0
- but still available for backwards compatibility
- Otherwise, consider using the directives that have replaced them.

8.4.1 PerlHandler

PerlHandler => PerlResponseHandler

8.4.2 PerlSendHeader

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

8.4.3 PerlSetupEnv

```
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.

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
- In mod_perl 2.0 a full tear-down and startup of interpreters is done on restart.
- To reuse the same *httpd.conf* for 1.0 and 2.0:

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

8.5 Code Porting

- Certain APIs couldn’t be preserved in 2.0, hence:

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

- which is slower, because it’s implemented in Perl.
- Consider getting rid of using the compat layer

- Remember the installation?

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

- Now mod_perl 1.0 and 2.0 can co-exist on the same machine
- mod_perl 2.0 modules go into the dir *Apache2/*

```
use Apache2;
```

- looks for path(s) with *Apache2/* using `@INC`
- prepends them to `@INC`
- `mp2doc` replaces `perldoc` to find mp2 manpages

8.5.1 Finding Which Modules Need To Be Loaded

The Poison:

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

- the execution fails with:

```
can't find method 'content_type' ...
```

- One has to load modules containing mod_perl methods before they can be used

The Remedy:

```
% perl -MApache2 -MModPerl::MethodLookup -e print_method content_type
to use method 'content_type' add:
         use Apache::RequestRec ();
```

Alias:

```
# c-style shell
% alias lookup "perl -MApache2 -MModPerl::MethodLookup -e print_method"

# bash-style shell
% alias lookup="perl -MApache2 -MModPerl::MethodLookup -e print_method"

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

8.6 ModPerl::Registry Family

- s/Apache::Registry/ModPerl::Registry/

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

- Issues with chdir(), which affects all threads
- MPM doesn’t matter, should work the same everywhere

- Cook your own registry with `ModPerl::RegistryCooker`
  - is-a: Subclass one of the classes
  - has-a: Build your own using aliasing

8.7 Method Handlers

- $$ doesn't work since some callbacks accepts more than 2 args

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

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

- See the *attributes* manpage.

8.8 Apache::StatINC Replacement

- Replace:

```
PerlInitHandler Apache::StatINC
```

- with:

```
PerlInitHandler Apache::Reload
```

- Apache::Reload provides an extra functionality
- See the module's manpage.

9 That’s all folks!

9.1 Thanks

Thanks to TicketMaster for sponsoring some of my work on mod_perl

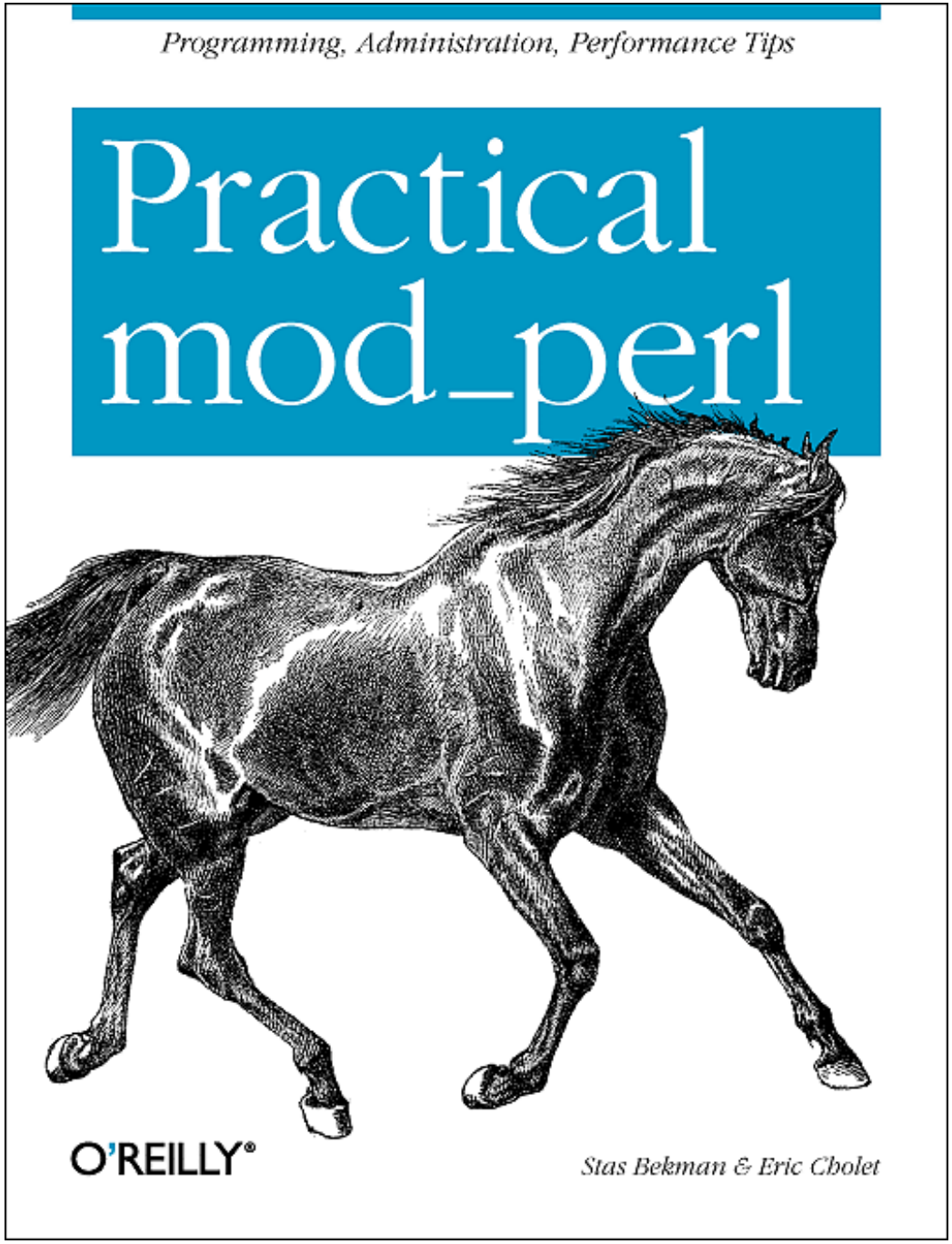
9.2 References

- mod_perl 2.0 information can be found at:

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

- Further Questions?
  - Grab me at the corridor and demand answers
  - Ask at modperl@perl.apache.org

9.3 A Shameless Plug


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