Object-Oriented mod_perl

Geoffrey Young
geoff@modperlcookbook.org
Overview

- Object-Oriented Perl Mechanics
- `mod_perl` Method Handlers
- Extending Core `mod_perl` Classes
Objects, smobjects

- Object-oriented programming has many advantages
  - subject of great debate
- Programming mod_perl in OO makes all kinds of wizardy possible
  - also makes julienne fries

- Let's have some fun using OOP
Perl OO Primer

- Some basic object-oriented features to understand
  - classes
  - methods
  - objects
  - inheritance
Pay Homage

- The entire object-oriented Perl world owes Damian Conway a huge debt of gratitude.
- The definitions that follow are essentially his...
- Any mistakes are unquestionably mine.
Perl Classes

- "To create a class, build a package"
- Perl packages associate variables and subroutines together under a common namespace

```perl
package My::Dinghy;

use 5.006;
use strict;

1;
```
Perl Methods

• "To create a method, build a subroutine"
• Perl subroutines can be called as functional subroutines
  
  \[
  \text{print } \$fh \ 'print() \ is \ a \ function';
  \]

• or as methods

  \[
  \$fh->\text{print('print() \ is \ a \ method');}
  \]

  \[
  \text{sub \ print \ { \}
  \phantom{.}my \ ($self, @data) = @_;}
  \]

http://www.modperlcookbook.org/
Perl Objects

• "To create an object, bless a referent"
• Perl has a special function `bless()` that associates a variable with a class

```perl
my $self = {}; return bless $self, $class;
```

• It's the variable that is associated with the class, not the reference to the variable
Perl Inheritance

- To create a subclass, populate @ISA
  - I made that up for consistency
- @ISA controls how Perl searches for methods when it can't find any in the subclass
- @ISA is a package global
  
  ```perl
  our @ISA = qw(My::Dinghy);
  use vars qw(@ISA);
  @ISA = qw(My::Dinghy);
  @My::12Meter::ISA = qw(My::Dinghy);
  ```
Guess What?

- mod_perl has already introduced you to most of Perl's OO semantics
  ```perl
  my $r = Apache->request;
  my $host = $r->headers_in->get('Host');
  ```
- In fact, mod_perl almost begs you to use OO
Handlers as Classes

package Cookbook::TrapNoHost;

use Apache::Constants qw(DECLINED BAD_REQUEST);
use Apache::URI;

use strict;

sub handler {
    my $r = shift;

    unless ($r->headers_in->get('Host') || $r->parsed_uri->hostname) {
        $r->custom_response(BAD_REQUEST,
                             "Oops!  Did you mean to omit a Host header?\n");
        return BAD_REQUEST;
    }
    return DECLINED;
};

1;

http://www.modperlcookbook.org/
OO mod_perl

• Programming using the mod_perl API forces us to use most of Perl's OO tools already

• We just need to fill in a few of the gaps for phenomenal cosmic power...
Step #1

- Change our existing handlers to method handlers
- Method handlers are just normal handlers called using OO syntax
- Allow us to use OO techniques to our advantage
Prototyping

- The classical way is to use Perl prototypes
  ```perl
  sub handler ($$$) {
      ...
  }
  ```
- Prototypes are deprecated in 2.0
Attributes

- The new way is to use subroutine attributes
  
  ```perl
  sub handler : method {
    ...
  }
  ```

- See the `attributes` manpage
Step #2

- Change our `handler()` method to be able to receive an OO call

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

- `$self` is the invoking class
  - most of the time
- `$r` is the same old Apache request object
Step #3

• Call the handler using a method syntax
  PerlModule My::MethodHandler
  PerlInitHandler My::MethodHandler->handler

• Pre-loading is required

• The arrow syntax is not
So What?

- Normal handlers and method handlers are equivalent in nearly all areas...
- ... but now you have the ability to inherit from other classes using OO techniques
For Example

- **Apache::SSI** provides a Perl implementation of Server Side Includes

  <Files * .shtml>
  SetHandler perl-script
  PerlHandler Apache::SSI
  </Files>

- Equivalent to mod_include except it adds a few important features...
Apache::SSI

• Integrates with Apache::Filter to provide filtered content generation

```xml
<Location /pipeline>
  SetHandler perl-script
  PerlHandler My::Content Apache::SSI Apache::Clean
  PerlSetVar Filter On
</Location>
```

• Pipelining like this is impossible using mod_cgi and mod_include
  – in Apache 1.3 at least
Drawbacks

• Apache::SSI is a huge win for people who like to modularize processing

• There is one rather limiting drawback to the current implementation

• If you use the exec or include SSI tag Apache::SSI must be the final filter in the chain

PerlHandler My::Content Apache::SSI

– due to implementation constraints
There is Hope

• Fortunately, Apache::SSI is implemented using method handlers
• We can subclass Apache::SSI and provide our own exec and include implementations that fix the problem
• We leave all the document parsing and other tag implementations alone

* Apache::SSI now includes Apache::FakeSSI which accomplishes almost the same thing
package Cookbook::SSI;

use Apache::SSI;
use HTTP::Request;
use LWP::UserAgent;
use strict;

@Cookbook::SSI::ISA = qw(Apache::SSI);

sub ssi_include {
    my ($self, $args) = @_;

    return $self->error("Include must be of type 'virtual'") unless $args->{virtual};

    my $uri = Apache::URI->parse(Apache->request);

    if ($args->{virtual} =~ m!^/!) {
        $uri->path($args->{virtual});         # path is absolute
    } else {
        my ($base) = $uri->path =~ m!(.*/)!;  # path is relative
        $uri->path($base . $args->{virtual});
    }

    my $request = HTTP::Request->new(GET => $uri->unparse);
    my $response = LWP::UserAgent->new->request($request);

    return $self->error("Could not Include virtual URL") unless $response->is_success;

    return $response->content;
}
1;

http://www.modperlcookbook.org/
Setup

• Just use our module wherever we used to use Apache::SSI

PerlModule Cookbook::SSI

<Location /pipeline>
  SetHandler perl-script
  PerlHandler My::Content Cookbook::SSI Apache::Clean PerlSetVar Filter On
</Location>

• the Apache::SSI engine takes care of everything but our new

  <!--#include virtual="/foo.pl" --> implementation
But wait, there's more...

- Method handlers are a nice thing to have
- Not very interesting in themselves
- Overriding core mod_perl classes is where the real fun begins
The Apache Class

- The Apache class is at the heart of all we do in mod_perl.
- Implements most of the amazing things we associate with the mod_perl API.
- You can make mod_perl do your own evil bidding by extending and overriding Apache.
Subclassing Apache

- Let's make `$r->bytes_sent()` return KB instead of bytes
- How? Create a simple subclass that does the calculation for us
package Cookbook::Apache;

use Apache;

use strict;

@Cookbook::Apache::ISA = qw(Apache);

sub new {
  my ($class, $r) = @_;

  $r ||= Apache->request;

  return bless { r => $r }, $class;
}

sub bytes_sent {
  return sprintf("%.0f", shift->SUPER::bytes_sent / 1024);
}

1;
H'wa?

• What's going on here?
  
  our @ISA = qw(Apache);
  return bless { r => $r }, $class;
  
  – ask Doug if you see him
  – typemap
    
    r = sv2request_rec($arg, "\$ntype\", cv)
  
  – sv2request_rec
    
    checks %$arg for _r or r keys
    calls sv2request_rec using _r or r for $arg

• Hey, it works
Sample Usage

```perl
package My::Bytes;

use Apache::Constants qw(OK);

use Cookbook::Apache;

use strict;

sub handler {
    my $r = shift;
    my $c = Cookbook::Apache->new($r);

    $c->log_error($c->bytes_sent, ' KB sent for ', $c->uri);
    $r->log_error($r->bytes_sent, ' bytes sent for ', $r->uri);

    return OK;
}
```

http://www.modperlcookbook.org/
Let's Simplify

- We only used both \$r and \$c in this example to show the difference
- Most of the time, you only need one request object, since your subclass inherits all the normal Apache methods

```perl
sub handler { 
  my \$r = Cookbook::Apache->new(shift);
}
```
Kick it up a notch

• Our sample *Apache* subclass isn't terribly interesting or terribly useful
• Time to add a little heat
Cross-Site Scripting

- As web developers, we should always check end-user input before using it.
- For `system()` calls, that means making sure no input is tainted:
  - running with `-T` switch
  - `PerlTaintCheck On`
- For HTML output, that means escaping input (`<` to `&lt;`) before displaying:
  - `HTML::Entities::escape()`
  - `Apache::Util::escape_html()`
Lots of Overhead

• For CGI or mod_perl developers, that means remembering lots of calls to `encode()` or `escape_html()`
  – which, of course, everyone does. Right?
• By subclassing the Apache class we can escape output automatically
• We don't want to escape all output, just the stuff from the end-user
• Get help from Taint.pm
Background

• We need to add `PerlTaintCheck On` to our `httpd.conf` to mimic the `-T` switch.

• Use `Taint:::tainted()` to determine whether data is tainted:

```perl
not eval { join('',@_), kill 0; 1 };```

• "send the data along"
package Cookbook::TaintRequest;

use Apache;
use Apache::Util qw(escape_html);

# Module load will die if PerlTaintCheck Off
use Taint qw(tainted);

use strict;

@Cookbook::TaintRequest::ISA = qw(Apache);

sub print {
    my ($self, @data) = @_;

    foreach my $value (@data) {
        # Dereference scalar references.
        $value = $$value if ref $value eq 'SCALAR';

        # Escape any HTML content if the data is tainted.
        $value = escape_html($value) if tainted($value);
    }

    $self->SUPER::print(@data);
}

http://www.modperlcookbook.org/
What's missing?

- So far all we have done is override mod_perl's `$r->print()`
- We still have to create a constructor for our class
- But what about just plain `print()`?
  ```perl
  print "<script>Heh!</script>"
  ```
- Our constructor needs to be special
package My::Tie;

sub TIEHANDLE {
    return bless {}, shift;
}

sub READLINE {
    return "reading...
";
}

tie *HANDLE, My::Tie;

while (my $line = <HANDLE>) {
    print $line;
}
The TIEHANDLE Interface

- Perl provides the TIEHANDLE interface as a way to override how filehandles behave when written to or read from

  ```perl
  print "foo";
  print STDOUT "foo";
  ```

- mod_perl uses tied filehandles to our advantage
More mod_perl Magic

- `$r->print()` sends data to the client
  - wrapper around Apache API calls that write data over the wire

- mod_perl ties standard streams to the Apache class
  - writes to `STDOUT` use `$r->print()`
    - why `Apache::Registry` works
  - reads from `STDIN` use Apache API calls to get data from the wire
Back to our example...

- So far we subclassed the Apache class to override mod_perl's `$r->print()`
- We still have to create a constructor for our class
- We need to intercept both `$r->print()` and `print()`s to STDOUT
The Apache class already provides a TIEHANDLE interface.

When subclassing the Apache class, we can override the PRINT part of the interface.

Leave the other TIEHANDLE parts in tact.
sub new {

    my ($class, $r) = @_; 

    $r ||= Apache->request;

    tie *STDOUT, $class, $r;

    return tied *STDOUT;
}
package Cookbook::TaintTest;

use Apache::Constants qw(OK);

use Cookbook::TaintRequest;

use strict;

sub handler {

    my $r = Cookbook::TaintRequest->new(shift);

    my @data = $r->args;

    # Untaint input data if magic word "override" is present.
    $data[1] =~ m/(.*override.*)/;
    $data[1] = $1 if $1;

    $r->send_http_header('text/html');
    $r->print("<html>You entered ", @data, "<br/></html>");

    return OK;
}

1;

http://www.modperlcookbook.org/
/tainted?x=\<script>alert("Hi!")\</script>

You entered x\<script>alert("Hi!")\</script>
/tainted?x=<script>alert("override Hi!")</script>
Registry has issues...

- Apache::Registry is incredibly cool
- But it's not perfect
  Apache::Registry - Run unaltered CGI scrips under mod_perl
- Scripts that work perfectly well under mod_cgi balk with Apache::Registry
  - that's why there's Apache::PerlRun
- Not even PerlRun can save you from some of the issues
For instance...

- Mixing mod_perl with legacy CGI code can be difficult
  - now you can isolate processing in phases other than content-generation

- What happens when you need POST data in other phases and legacy Apache::Registry code?

- What we need is a way to cache POST data that makes it accessible to legacy CGI scripts
Registry: The Next Generation

- Apache::Registry is pretty complex and difficult to alter
- Apache::RegistryNG behaves almost the same, but is object-oriented and a much better candidate for subclassing

- Apache::RegistryNG is actually a subclass of Apache::PerlRun
More Subclasses!

- Let's fix the problem by subclassing `Apache::RegistryNG`
- `tie STDIN instead...`
package Apache::CachePOSTRegistry;

use Apache::RegistryNG;
use Apache::Request;
use strict;

@Apache::CachePOSTRegistry::ISA = qw(Apache::RegistryNG);

sub new {
    my ($class, $r) = @_; 

    $r = Apache::Request->instance($r || Apache->request);

    tie *STDIN, $class, $r;

    return tied *STDIN;
}

sub TIEHANDLE {
    my ($class, $r) = @_; 

    return bless { r => $r }, $class;
}
Apache::Request->instance()

- Same as Apache::Request->new()
- Stores away an Apache::Request object (with the POST data) and uses it for future instantiations

```perl
sub instance {
    my $class = shift;
    my $r = shift;
    if (my $apreq = $r->pnotes('apreq')) {
        return $apreq;
    }
    my $new_req = $class->new($r, @_);
    $r->pnotes('apreq', $new_req);
    return $new_req;
}
```
sub READ {
    my $self = shift;
    my $buf = \$_[0]; shift;
    my $len = shift;
    my $offset = shift || 0;

    my @args = ();

    $self->{r}->param->do(sub {
        push @args, join '=', @_;  
        1;
    });

    my $input = join '&', @args;
    $input =~ s! !+!g;

    substr($$buf, $offset) = substr($input, 0, $len);
    substr($input, 0, $len) = '';
    return length substr($$buf, $offset);
}

1;
Alias /perl-bin /usr/local/apache/perl-bin

<Location /perl-bin>
  SetHandler perl-script
  PerlHandler Apache::Registry
  Options +ExecCGI
  PerlSendHeader On
</Location>
It Works!

sub My::InitHandler {
    my $r = Apache::Request->instance(shift);
    my @post = $r->param;
    ...
    return Apache::Constants::OK;
}
1;

#!/usr/bin/perl

read(STDIN, my $posted, $ENV{'CONTENT_LENGTH'});

print "Content-type: text/plain\n\n";
print $posted;
It Works!

#!/usr/bin/perl
use CGI;
my $q = CGI->new;
print $q->header(-type=>'text/plain');
print $q->param;
XS Anyone?

- Just for kicks, let's throw some XS into the mix
- XS is the glue that ties Apache and Perl together
  - most of the mod_perl API lives in Apache.xs
- Unfortunately, mod_perl 1.3 doesn't offer up all the Apache API, only selected parts
- With XS, we can open up the rest...
Assbackwards

- In Apache-speak, a request that is assbackwards is an HTTP/0.9 request
- Apache still supports HTTP/0.9

$ telnet localhost 80
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
GET /cgi-bin/sayhello.cgi
The assbackwards Flag

- Apache marks HTTP/0.9 requests with the assbackwards flag in the request record
- If $r->assbackwards is set, Apache doesn't send any headers
- mod_perl does not provide a way to access the assbackwards flag
- but it uses it when convenient
  my $sub = $r->lookup_uri('/layline.html');
  $sub->run(1);
- You can too!
Approach

• Accessing parts of the Apache API that mod_perl does not natively support requires XS
  - Assbackwards.pm
  - Assbackwards.xs
  - typemap
  - Makefile.PL

• Fortunately, in this case, all the parts are simple
The Perl Part
package Apache::Assbackwards;

use 5.006;
use strict;
use warnings;

use DynaLoader;

our @ISA = qw(DynaLoader Apache);

our $VERSION = '0.01';

__PACKAGE__->bootstrap($VERSION);

sub new {
  my ($class, $r) = @_;

  $r ||= Apache->request;

  return bless { r => $r }, $class;
}
1;

http://www.modperlcookbook.org/
The XS Part
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
#include "mod_perl.h"
#include "mod_perl_xs.h"

MODULE = Apache::Assbackwards PACKAGE = Apache::Assbackwards

PROTOTYPES: ENABLE

int
assbackwards(r, ...)
   Apache r

   CODE:
       get_set_IV(r->assbackwards);

   OUTPUT:
       RETVAL
The typemap Part

TYPEMAP
Apache T_APACHEOBJ

OUTPUT
T_APACHEOBJ
   sv_setref_pv($arg, \"${ntype}\", (void*)$var);

INPUT
T_APACHEOBJ
   r = sv2request_rec($arg, \"${ntype}\", cv)
use Apache::src ();

WriteMakefile(
    NAME => 'Apache::Assbackwards',
    VERSION_FROM => 'Assbackwards.pm',
    INC => Apache::src->new->inc,
}

http://www.modperlcookbook.org/
Just use the canonical

$ perl Makefile.PL
$ make
$ sudo make install

and you're good to go
How do we use it?

```perl
package My::Assbackwards;

use Apache::Assbackwards;
use strict;

sub handler {
    my $r = Apache::Assbackwards->new(shift);

    $r->assbackwards(1);

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

http://www.modperlcookbook.org/
$ telnet localhost 80
Trying 127.0.0.1...
Connected to localhost.
Escape character is '\[\].
GET /cgi-bin/sayhello.cgi HTTP/1.0

HTTP/1.1 200 OK
Date: Sat, 15 Jun 2002 19:08:48 GMT
Server: Apache/1.3.25-dev (Unix) mod_perl/1.27_01-dev Perl/v5.8.0
Expires: Sat, 15 Jun 2002 19:08:50 GMT
Connection: close
Content-Type: text/plain; charset=ISO-8859-1

Hi

$ telnet localhost 80
Trying 127.0.0.1...
Connected to localhost.
Escape character is '\[\].
GET /cgi-bin/sayhello.cgi HTTP/1.0

Hi
That was fun, but...

- Ok, the assbackwards flag is amusing but not all too practical
- How about something more applicable?
  - ap_note_digest_auth_failure
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
#include "mod_perl.h"

MODULE = Apache::AuthDigest::API
PACKAGE = Apache::AuthDigest::API

PROTOTYPES: ENABLE

void
note_digest_auth_failure(r)
    Apache r

    CODE:
    ap_note_digest_auth_failure(r);
... 

our $VERSION = '0.01';
our @ISA = qw(DynaLoader Apache);

__PACKAGE__->bootstrap($VERSION);

...

sub new {
  my ($class, $r) = @_; 

  $r ||= Apache->request;

  return bless { r => $r }, $class;
}

...

unless ($response) {
  $log->info("Client did not supply a Digest response");

  $r->note_digest_auth_failure;
  return AUTH_REQUIRED
}
More?

- Apache provides for something called "registered cleanups"
  
  $r->register_cleanup([&cleanup]);
  
  PerlCleanupHandler My::Cleanup
  
  Apache->server->register_cleanup([&cleanup]);

- Cleanups allow you to schedule processing at various end points in the Apache runtime

- All use Apache's concept of memory pools
Keepalives

- **HTTP/1.1** allows for more than one request per connection
  
  ```perl
  my $c = $r->connection;
  
  connection: Keep-Alive
  
  - Connections are interfaces with the Apache::Connection class
    
    ```perl
    my $c = $r->connection;
    ```

  - Apache creates a per-connection memory pool

  - **mod_perl** provides no way to register per-connection cleanups
Per-connection Cleanups

- Unfortunately, mod_perl wasn't designed so you could subclass the Apache::Connection class
- Let's subclass the Apache class and re-bless $r->connection into our class
package Apache::ConnectionCleanup;

use 5.006;
use strict;

use Apache;
use Apache::ConnectionCleanup::RegisterCleanup;

our @ISA = qw(Apache);

sub new {
    my ($class, $r) = @_;

    $r ||= Apache->request;

    return bless { r => $r }, $class;
}

sub connection {
    my $connection = shift->SUPER::connection;

    return bless $connection,
        'Apache::ConnectionCleanup::RegisterCleanup';
}

1;

http://www.modperlcocookbook.org/
package Apache::ConnectionCleanup::RegisterCleanup;

use 5.006;
use strict;
use warnings;

use Apache::Connection;
use DynaLoader;

our @ISA = qw(DynaLoader Apache::Connection);

our $VERSION = '0.01';

__PACKAGE__->bootstrap($VERSION);

1;
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
#include "mod_perl.h"

...static void ApacheConnection_register_cleanup(conn_rec *c, SV *cv) {
    pool *p = c->pool;

    ...

    register_cleanup(p, conn, conn_cleanup_handler, mod_perl_noop);
}

...

void
register_cleanup(conn, cv)
    Apache::Connection conn
    SV *cv

    CODE:
    ApacheConnection_register_cleanup(conn, cv);
How Do We Use It?
use Apache::ConnectionCleanup;

use strict;

sub handler {
    my $r = Apache::ConnectionCleanup->new(shift);
    $r->connection->register_cleanup(&cleanup);
    return OK;
}

sub cleanup {
    # do something
}

Problems?

• This particular interface presents an interesting problem
  – the cleanup is added by getting the connection from the request record
  – multiple requests are involved in KeepAlive connections
  – so, we end up registering multiple connection cleanups

• This may or may not be a problem for you…
One way around it...

- There is a solution
  - tricky one
  - probably not the only one
  - probably not the best one
  - but it works
package My::Single;

use Apache::Constants qw(OK);
use Apache::ConnectionCleanup;
use strict;

$My::Single::Run = 0;

sub handler {
    my $r = Apache::ConnectionCleanup->new(shift);

    # increment the counter for each request
    $My::Single::Run++;

    # and create a closure to reset the counter after the first run
    my $marker = bless {}, 'My::Single';

    $r->connection->register_cleanup(sub { single($marker) });

    return OK;
}

http://www.modperlcookbook.org/
sub single {
    my $marker = shift;

    # only run if we're allowed
    return unless $My::Single::Run;

    # do stuff...
}

sub DESTROY {
    # after the first closure is complete, reset the
    # counter to make sure we only run once
    $My::Single::Run = 0;
}

Fine Manuals

- **Writing Apache Modules with Perl and C**
  - http://www.modperl.com/

- **mod_perl Developer's Cookbook**
  - http://www.modperlcookbook.org/

- **mod_perl Pocket Reference**
  - http://www.refcards.com/

- **mod_perl Guide**
  - http://perl.apache.org/guide/
  - http://www.modperlbook.org/

- **mod_perl at the ASF**
  - http://perl.apache.org/
Materials

These slides
http://www.modperlcookbook.org/~geoff/slides/ApacheCon

My modules
http://www.modperlcookbook.org/~geoff/modules