Interfacing Apache HTTP Server 2.4 with External Applications

Jeff Trawick

November 6, 2012

Who am I?

- Met Unix (in the form of Xenix) in 1985
- Joined IBM in 1990 to work on network software for mainframes
- Moved to a different organization in 2000 to work on Apache httpd
- Later spent about 4 years at Sun/Oracle
- Got tired of being tired of being an employee of too-huge corporation so formed my own too-small company
- Currently working part-time, coding on other projects, and taking classes

Overview

- Huge problem space, so simplify
- Perspective: "General purpose" web servers, not minimal application containers which implement HTTP
- "Applications:" Code that runs dynamically on the server during request processing to process input and generate output

Possible web server interactions

- Native code plugin modules (uhh, assuming server is native code)
- Non-native code + language interpreter inside server (Lua, Perl, etc.)
- Arbitrary processes on the other side of a standard wire protocol like HTTP (proxy), CGI, FastCGI, etc. (Java and "all of the above") or private protocol
- Some hybrid such as mod_fcgid

mod_fcgid as example hybrid

- Supports applications which implement a standard wire protocol, no restriction on implementation mechanism
- Has extensive support for managing the application[+interpreter] processes so that the management of the application processes is well-integrated with the web server

Contrast with mod_proxy_fcgi (pure FastCGI, no process management) or mod_php (no processes/threads other than those of web server).

Application space requirements

- native code plugin module — understand at least some of the internal request processing phases, take control of certain phases
- external processes — implement a protocol to communicate with the web server
    - libraries already exist for standard protocols (HTTP, CGI, FastCGI, etc.), although in some cases the protocol is trivial to implement with no help

APIs

- may mirror web server API (like mod_perl)
- may be more generic like the servlet API
- non-API: just run and generate output

```
<?php
  echo "Hello, world!";
?>
```

Native module drawbacks

- Overall resource use often larger when app runs in the web server, especially for prefork model
  - memory
  - connections to database, LDAP, etc.

  Resources are often left behind on any thread/process that occasionally runs the application — underutilized.
- Introduces instability into server
- Collisions between requirements of different modules
- Generally unable to support multiple script interpreter versions
- Potential lack of thread safety, or expensive locking

But by running applications in their own processes

- Often the required application thread/process count is a fraction of that of the web server (so resources not left behind on threads/processes occasionally used).
- A particular application usually can't introduce instability into the server, so basic services and other applications are unaffected.
- Different applications can use different libraries, interpreter versions, framework versions, etc.
- Independent start/stop of web server and application
- Independent identity or chroot env vs. web server and other applications

Where are we now with app containers?

- larger numbers of popular server implementations (*market fragmentation*)
- anywhere from using script interpreter CLI to invoke mini HTTP engine, IDE-controlled servers for development, traditional "web servers" like httpd & nginx (mod_foo? CGI? FastCGI?) to cloud deployment on Heroku, App Engine, etc. with hidden implementation
- lots of implementations/protocols/APIs
  - to choose from as an app developer
  - to need to support as a hopeful framework provider

Solution: separate the interface from the implementation

- application space
  - both the quick-and-dirty-script writer as well as the framework provider write to a sanitized API instead of to different transport or web server APIs
    - (or to a collection of different APIs on the part of the framework provider)
- run-time provider
  - (service provider, server provider, third-party *glue* provider) makes the sanitized API work on their run-time environment, and doesn't need to get the different types of developers to target their run-time

But look at CGI.pm as an obvious (and old) example:

- CGI
- FastCGI
- mod_perl
- even a couple of ways to map CGI.pm to PSGI

That's plenty portable among possible run-time environments.

Limiting the scope of examples in this presentation

- simple application
- use the sanitized APIs for four popular scripting languages: Perl, PHP, Python, and Ruby
- forget about HTTP proxy to other run-time environments or anything Java

command-line script versions — Perl

```
#!/usr/bin/env perl
use DBI;
use Cache::Memcached;
my $key = 'Monday';
my $mckey = 'demowebapp.' . $key;
my $mc = new Cache::Memcached({'servers' => ['192.168.11.199:11211']});
my $val = $mc->get($mckey);
if (!$val) {
    my $dbh = DBI->connect('DBI:Pg:dbname=demowebapp;host=192.168.11.199');
    my $sth = $dbh->prepare("SELECT * FROM demowebapp_x WHERE id = '$key';");
    $sth->execute();
    ($key, $val) = $sth->fetchrow_array();
    $sth->finish();
    $dbh->disconnect();
    $mc->set($mckey, $val, 1);
}
print "$val\n";
```

command-line script versions — PHP

```
#!/usr/bin/env php
<?php
$key = 'Monday';
$mckey = 'demowebapp.' . $key;
$mc = new Memcache;
$mc->connect('192.168.11.199', 11211);
$val = $mc->get($mckey);
if (!$val) {
    $pgconn = pg_connect("host=192.168.11.199 dbname=demowebapp");
    $res = pg_query($pgconn, "SELECT * from demowebapp_x WHERE id = '$key';");
    $row = pg_fetch_row($res);
    $val = $row[1];
    pg_free_result($res);
    pg_close($pgconn);
    $mc->set($mckey, $val, 0, 1);
}
print "$val\n";
?>
```

command-line script versions — Python

```
#!/usr/bin/env python
import psycopg2
import memcache
key = 'Monday'
mckey = 'demowebapp.' + key
mc = memcache.Client(['192.168.11.199:11211'])
val = mc.get(mckey)
if not val:
    pg = psycopg2.connect(database='demowebapp', host='192.168.11.199')
    csr = pg.cursor()
    csr.execute("SELECT * FROM demowebapp_x WHERE id = '%s';" % key)
    val = csr.fetchone()[1]
    csr.close()
    pg.close()
    mc.set(mckey, val, time=1)
print val
```

command-line script versions — Ruby

```
#!/usr/bin/env ruby
require 'memcached'
require 'pg'
key = 'Monday'
mckey = 'demowebapp.' + key
mc = Memcached.new('192.168.11.199:11211')
begin
  val = mc.get mckey, false
rescue
  val = nil
end
if not val
    pgconn = PGconn.open(:dbname => 'demowebapp', :host => '192.168.11.199')
    res = pgconn.exec("SELECT * from demowebapp_x WHERE id = '#{key}';")
    val = res[0]['content']
    res.clear
    pgconn.finish
    mc.set mckey, val, 1, false
end
print "#{val}\n"
```

Web APIs

- PSGI for Perl
- Rack for Ruby
- WSGI for Python
- PHP? *same old same old*

Commonalities between PSGI, Rack, and WSGI

The basic API description is the same.

- Input:
  - CGI-like variables, input handle for request body if necessary, handle for error messages, information about the run-time environment
- Output:
  - A structure with the HTTP response code, response headers, and either the response body or some readable object
- PSGI and Rack based on WSGI *but most of this is the essence of the problem anyway*

PSGI — Perl Web Server Gateway Interface

- Interface between applications (or frameworks) and run-time environment
- A PSGI app is a Perl subroutine which adheres to this minimal spec
- Plack is a set of adapters to web servers (”PSGI Toolkit”)
  - CGI, SCGI, FastCGI, mod_perl, *more*
  - Plack::Handler::Apache2, Plack::Handler::FCGI, Plack::Handler::CGI, etc.
- Other providers besides Plack

WSGI — Web Server Gateway Interface

Web Server Gateway Interface for Python

- Supported by lots of frameworks
- FastCGI, CGI, SCGI, mod_wsgi, more

Rack

Rack interface for Ruby

- Supported by lots of frameworks
- Rackup, Phusion Passenger

PHP

- embedded in HTML or not, the model is the same (though some capabilities differ by run-time environment)

Webapp versions in Perl, PHP, Python, and Ruby

These are also at `http://emptyhammock.blogspot.com/2012/11/app-app-app-app.html`.

Webapp version — Perl

```
use strict;
use DBI;
use Cache::Memcached;

sub get_key {
    my $pi = shift;
    my @terms = split(/\//, $pi || "Monday");
    return $terms[-1];
}

my $app = sub {
    my $env = shift;

    my $key = get_key($env->{'PATH_INFO'});
    my $mckey = 'demowebapp.' . $key;

    my $mc = new Cache::Memcached({'servers' => ['192.168.11.199:11211']});
    my $val = $mc->get($mckey);
    if (!$val) {
        my $dbh = DBI->connect('DBI:Pg:dbname=demowebapp;host=192.168.11.199');
        my $sth = $dbh->prepare("SELECT * FROM demowebapp_x WHERE id = '$key';");
        $sth->execute();
        ($key, $val) = $sth->fetchrow_array();
        $sth->finish();
        $dbh->disconnect();
        $mc->set($mckey, $val, 1);
    }

    return ['200', ['Content-Type' => 'text/html'], [$val]];
};
```

Webapp version — PHP

```
<?php

function get_key($pi) {
    $terms = strlen($pi) != 0 ? $pi : 'Monday';
    $key = end(explode('/', $terms));
    return $key;
}

$key = get_key($_SERVER['PATH_INFO']);
$mckey = 'demowebapp.' . $key;

$mc = new Memcache;
$mc->connect('192.168.11.199', 11211);
$val = $mc->get($mckey);
if (!$val) {
    $pgconn = pg_connect("host=192.168.11.199 dbname=demowebapp");
    $res = pg_query($pgconn, "SELECT * from demowebapp_x WHERE id = '$key';");
    $row = pg_fetch_row($res);
    $val = $row[1];
    pg_free_result($res);
    pg_close($pgconn);
    $mc->set($mckey, $val, 0, 1);
}

print "$val\n";
?>
```

Webapp version — Python

```
import psycopg2
import memcache

def get_key(pi):
    terms = [token for token in pi.split('/') if token != '']
    if terms:
        return terms[-1]
    return 'Monday'

def application(environ, start_response):
    start_response('200 OK', [('Content-type', 'text/html')])
    key = get_key(environ['PATH_INFO'])
    mckey = 'demowebapp.' + key
    mc = memcache.Client(['192.168.11.199:11211'])
    val = mc.get(mckey)
    if not val:
        pg = psycopg2.connect(database='demowebapp', host='192.168.11.199')
        csr = pg.cursor()
        csr.execute("SELECT * FROM demowebapp_x WHERE id = '%s';" % key)
        val = csr.fetchone()[1]
        csr.close()
        pg.close()
        mc.set(mckey, val, time=1)
    return [val]
```

Webapp version — Ruby

```
class Lookup
    def get_key(pi)
        terms = pi != nil ? pi : 'Monday'
        terms.split('/')[-1]
    end

    def call env
        key = get_key(env['PATH_INFO'])
        mckey = 'demowebapp.' + key

        mc = Memcached.new('192.168.11.199:11211')
        begin
            val = mc.get mckey, false
        rescue
            val = nil
        end
        if not val
            pgconn = PGconn.open(:dbname => 'demowebapp', :host => '192.168.11.199')
            res = pgconn.exec("SELECT * from demowebapp_x WHERE id = '#{key}';")
            val = res[0]['content']
            res.clear
            pgconn.finish
            mc.set mckey, val, 1, false
        end

        [200, {'Content-Type' => 'text/html'}, [val]]
    end
end
```

Generally, how to run with httpd

- script interpreter inside httpd, running on the request thread
- external process, new process for every request (CGI)
- pool of external process, ability to manage the pool (FastCGI and others), some form of IPC between request thread (handler) and external process

Simple deployment

Each of these languages and APIs have a simple run-time environment for use during development which allows you to start a minimal HTTP server from the command-line.

PSGI

- Easiest container for dev is HTTP::Server::PSGI

```
$ plackup -e 'sub { [200, ["Content-Type" => "text/plain"], ["Hello, world!"]] }'
HTTP::Server::PSGI: Accepting connections at http://0:5000/
```

  (from `http://en.wikipedia.org/wiki/Plack_(software)`)
- has automatic reload capability
- similar, but via FastCGI

```
$ plackup -s FCGI --listen /tmp/fcgi.sock -e 'sub { [200, ["Content-Type" => "text/plain"
FastCGI: manager (pid 8315): initialized
FastCGI: manager (pid 8315): server (pid 8316) started
FastCGI: server (pid 8316): initialized
```

PHP

```
$ ~/php54inst/bin/php -S 127.0.0.1:9999 -c $HOME/php54inst/etc
PHP 5.4.8 Development Server started at Tue Nov  6 08:12:53 2012
Listening on http://127.0.0.1:9999
Document root is /home/trawick/myhg/apache/documents/AC2012EU
Press Ctrl-C to quit.
[Tue Nov  6 08:20:43 2012] 127.0.0.1:42825 [200]: /lookup.php
...
```

(new with PHP 5.4)

WSGI

```
$ uwsgi  --plugins http,python --http :9090 --wsgi-file ./lookup.wsgi
/usr/lib/uwsgi/plugins/python27_plugin.so
*** Starting uWSGI 0.9.8.1-debian (64bit) on [Tue Nov  6 09:00:45 2012] ***
compiled with version: 4.6.1 on 28 June 2011 10:48:13
 *** WARNING: you are running uWSGI without its master process manager ***
your memory page size is 4096 bytes
spawned uWSGI http 1 (pid: 4530)
HTTP router/proxy bound on :9090
...
```

Rack

- config.ru:

```
require 'memcached'
require 'pg'
require './lookup'
run Lookup.new
```

- command-line:

```
$ rackup config.ru
[2012-11-06 08:57:04] INFO  WEBrick 1.3.1
[2012-11-06 08:57:04] INFO  ruby 1.9.2 (2011-07-09) [x86_64-linux]
[2012-11-06 08:57:04] INFO  WEBrick::HTTPServer#start: pid=4435 port=9292
```

Proxy to command-line development environment?

mod_proxy could be used to route requests to these mini servers, but generally they are intended only for development (with the exception of uwsgi).

PSGI with httpd 2.4

- mod_cgi[d], mod_fcgid, mod_proxy_fcgi
- mod_perl
  - Umm, mod_perl for 2.4 is a work in progress; mod_perl exports the gory details of the module API, and that work isn't finished.
  - A step by step guide for mod_perl with httpd 2.4 is available, and PSGI shouldn't be impacted by the lingering issues, but YMMV.
- mod_psgi
  - Looks nice and small, appears to have limited use, referred to as *experimental* in some references...
  - Needs a patch to work with Perl <5.14
  - Nonetheless, the module builds fine with 2.4 (ignoring the Perl 5.14 dependency, which I didn't have).

WSGI with httpd 2.4

- mod_cgi[d], mod_fcgid, mod_proxy_fcgi
- mod_wsgi

(Err, Phusion Passenger has support for WSGI too, but mod_wsgi is the one.)

Rack with httpd 2.4

- mod_cgi[d], mod_fcgid, mod_proxy_fcgi
- Phusion Passenger (supports httpd 2.4 as of Phusion Passenger 3.0.2)

PHP with httpd 2.4

- mod_cgi[d], mod_fcgid, mod_proxy_fcgi
- mod_php
    - New with 5.4: `http://wiki.apache.org/httpd/PHP-FPM`

WHAT ABOUT SCGI?

- http://python.ca/scgi/protocol.txt
- http://en.wikipedia.org/wiki/Simple_Common_Gateway_Interface
- PHP doesn't support it (https://bugs.php.net/bug.php?id=36943), but PSGI, Rack, and WSGI apps can be accessed via SCGI with the proper container.
- Ignore for brevity.

Something else ignored intentionally: mod_fastcgi

PHP via mod_fcgid

- PHP FastCGI processes normally exit after 500 requests

  Synchronize mod_fcgid and PHP limits to avoid 500 error.

  In PHP wrapper:

  ```
  PHP_FCGI_MAX_REQUESTS=10000
  ```

  In fcgid configuration:

  ```
  FcgidMaxRequestsPerProcess 10000
  ```

  or just set PHP_FCGI_MAX_REQUESTS to 0 to disable

PHP — Special considerations with mod_fcgid

- PHP FastCGI process management ineffective (wasted) with mod_fcgid, which routes only single concurrent requests to the socket of a process which it has spawned.

  Leave PHP child process management disabled (PHP_FCGI_CHILDREN=0).

PHP — Special considerations with mod_fcgid

But:

- With PHP process management, single cache can be used concurrently by many processes.
- Without PHP child process management, PHP opcode caches are not very effective. Cache is serially reused within single process when the same fcgid-spawned process handles another request.

PHP — Perhaps unexpected issues when running as FastCGI

- PHP flags in .htaccess files — no longer respected when you move from mod_php to FastCGI
- on Windows, mod_php strips the drive letter from SCRIPT_NAME; mod_fcgid doesn't

PHP — Configuration

.conf:

```
LoadModule fcgid_module modules/mod_fcgid.so

FcgidMaxRequestsPerProcess 5000

# Uncomment the following line if cgi.fix_pathinfo is set to 1 in
# php.ini:
# FcgidFixPathinfo 1

Alias /php/ /home/trawick/myhg/apache/documents/AC2012EU/php/
<Directory /home/trawick/myhg/apache/documents/AC2012EU/php/>
Options +ExecCGI
AddHandler fcgid-script .php
FcgidWrapper /home/trawick/myhg/apache/documents/AC2012EU/ .php
# 2.4-specific
Require all granted
</Directory>
```

PHP — Configuration (cont.)

wrapper script:

```
#!/bin/sh
export PHPRC=/home/trawick/myhg/apache/documents/AC2012EU/
export PHP_FCGI_MAX_REQUESTS=5000
export PHP_FCGI_CHILDREN=8
exec /usr/bin/php-cgi
```

WSGI via mod_wsgi (internal)

```
LoadModule wsgi_module modules/mod_wsgi.so
WSGIScriptAlias /wsgi/ /home/trawick/myhg/apache/documents/AC2012EU/
<Directory /home/trawick/myhg/apache/documents/AC2012EU/>
# 2.4-specific
Require all granted
</Directory>
```

WSGI via mod_wsgi (external)

```
LoadModule wsgi_module modules/mod_wsgi.so
WSGIDaemonProcess test processes=2 threads=25
WSGIScriptAlias /wsgi/ /home/trawick/myhg/apache/documents/AC2012EU/
<Directory /home/trawick/myhg/apache/documents/AC2012EU/>
# 2.4-specific
Require all granted
</Directory>
```

WSGIDaemonProcess has a host of options, including

- run as a different user/group when starting httpd as root
- configure I/O timeouts and buffer sizes
- set display name for ps

`http://code.google.com/p/modwsgi/wiki/ConfigurationDirectives#WSGIDaemonProcess`

PSGI via mod_proxy_fcgi

Starting the FastCGI processes:

```
$ plackup -s FCGI --listen 127.0.0.1:10081 --daemonize --nproc 10 ./lookup.psgi
```

.conf:

```
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_fcgi_module modules/mod_proxy_fcgi.so
ProxyPass /psgi/ fcgi://127.0.0.1:10081/
```

Rack via Phusion Passenger

.conf:

```
# Note: The installer picks a Ruby, possibly not the one
# you want.
LoadModule passenger_module /var/lib/gems/1.8/gems/passenger-3.0.18/ext/apache2/mod_passenger.sc
PassengerRoot /var/lib/gems/1.8/gems/passenger-3.0.18
PassengerRuby /usr/bin/ruby1.8
Listen 8081
<VirtualHost *:8081>
    DocumentRoot /home/trawick/inst/24-64/htdocs/rackapps/lookup/public
    <Directory /home/trawick/inst/24-64/htdocs/rackapps/lookup/public>
        Require all granted
        Options -MultiViews
    </Directory>
</VirtualHost>
```

Other: In the `lookup` directory (referenced above), create `public` and `tmp` directories and store `config.ru` and `lookup.rb` under `lookup`. `config.ru` is unchanged from rackup command-line deployment.

Compared with httpd 2.2

- Just about everything in this space that works with 2.4 will work with 2.2.
- A couple of special issues to keep in mind:
  - mod_perl is still bleeding edge on 2.4 because of the way it exposes a rich set of httpd APIs and is affected by most any change anywhere, which isn't the general scenario.

  Summary: None of the unbundled solutions are bleeding edge on 2.2, but mod_perl is with 2.4.
  - mod_proxy_fcgi is not part of 2.2, though there is a third-party module by that name available for 2.2.

Compared with nginx 1.2.latest

Generalities:

- nginx doesn't do any process management
  - no CGI support at all
  - application processes not part of web server lifecycle
- Any potential mechanism for running scripts inside nginx will impose big limitations (don't block).

FastCGI differences with nginx

- No process management, so nothing like mod_fcgid (standard recommendation is to use `spawn-fcgi` from Lighttpd, though containers that provide a mapping of a standard API to FastCGI usually provide the same capability)
- FastCGI capability similar to mod_proxy_fcgi, but also supports Unix sockets (a patch surfaced recently to add Unix socket support to mod_proxy_fcgi)
- nginx (apparently) doesn't support load balancing to FastCGI (unlike mod_proxy_fcgi) but some FastCGI apps like PHP can spawn multiple children on the same socket in order to handle load balancing.
- mod_fcgid provides process managment, but has the reverse limitation: mod_fcgid will route requests only to processes it has spawned, and only one concurrent request per process.

Rack and nginx

- Phusion Passenger also supports nginx
- FastCGI, ...

WSGI and nginx

- experimental mod_wsgi-for-nginx
- ability to forward to uWSGI which supports WSGI and other protocols
- FastCGI, ...

PHP as FastCGI with nginx

- nginx doesn't handle process management, so use something else (`php-cgi -b BINDADDR` will work).
- Any *wrapper script* is of no interest to nginx, but it still is a good place to set up PHP with the desired settings.

```
#!/bin/sh
export PHP_FCGI_CHILDREN=20
export PHP_FCGI_MAX_REQUESTS=5000
/usr/bin/php-cgi -b 127.0.0.1:10080
```

PHP as FastCGI with nginx (cont.)

```
location ~ \.php$ {
  # fastcgi_params is part of standard configuration
  include fastcgi_params;
  fastcgi_pass 127.0.0.1:10080;
}
```

(Both php-cgi and fastcgi_pass support Unix sockets.)

PSGI

- http://en.wikipedia.org/wiki/PSGI/
- http://search.cpan.org/~miyagawa/PSGI-1.101/PSGI.pod
- http://www.catalyzed.org/2009/11/mtplack-on-nginx-love.html
- https://github.com/spiritloose/mod_psgi/
- http://www.simon-cozens.org/content/i-finally-get-psgi-and-plack
- http://www.reddit.com/r/perl/comments/h6qqr/the_psgi_is_the_limit/
- http://plackperl.org
- http://search.cpan.org/~miyagawa/Plack-1.0009/lib/Plack/Handler/CGI.pm
- http://en.wikipedia.org/wiki/Plack_(software)
- http://search.cpan.org/~miyagawa/PSGI-1.101/PSGI.pod http://www.gossamer-threads.com/lists/

WSGI

- http://www.python.org/dev/peps/pep-3333/
- http://stackoverflow.com/questions/2532477/mod-cgi-mod-fastcgi-mod-scgi-mod-wsgi-mod-python-
- http://stackoverflow.com/questions/219110/how-python-web-frameworks-wsgi-and-cgi-fit-togeth
- http://code.google.com/p/modwsgi/wiki/ConfigurationDirectives

Rack

- `http://en.wikipedia.org/wiki/Rack_(web_server_interface)`
- `http://m.onkey.org/ruby-on-rack-1-hello-rack`

FastCGI

- `http://people.apache.org/~trawick/AC2010-FastCGI.pdf`
- `http://httpd.apache.org/mod_fcgid/mod/mod_fcgid.html`