openresty / echo-nginx-module

An Nginx module for bringing the power of "echo", "sleep", "time" and more to Nginx's config file
http://wiki.nginx.org/NginxHttpEchoModule

426 commits | 2 branches | 75 releases | 3 contributors

branch: master echo-nginx-module / +

bumped version to 0.57.

agentzh authored on Nov 22, 2014 latest commit 91ee9a8230

doc	bumped version to 0.57.	3 months ago
src	bugfix: $echo_client_request_headers: buffer overflow and the "buffer…	5 months ago
t	bugfix: $echo_client_request_headers: buffer overflow and the "buffer…	5 months ago
util	util/build.sh: no longer add luajit lib to RPATH.	5 months ago
.gitignore	updated .gitignore a bit.	2 years ago
LICENSE	updated the LICENSE file to match README.	4 months ago
README.markdown	bumped version to 0.57.	3 months ago
config	various file name and coding style fixes.	5 years ago
valgrind.suppress	suppressed a valgrind false positive in libdl.	11 months ago

README.markdown

Name

ngx_echo - Brings "echo", "sleep", "time", "exec" and more shell-style goodies to Nginx config file.

This module is not distributed with the Nginx source. See the installation instructions.

Table of Contents

Status

This module is production ready.

Version

This document describes ngx_echo v0.57 released on 21 November 2014.

Synopsis

```
location /hello {
  echo "hello, world!";
}
```

```
location /hello {
  echo -n "hello, ";
  echo "world!";
}
```

```
location /timed_hello {
  echo_reset_timer;
  echo hello world;
  echo "'hello world' takes about $echo_timer_elapsed sec.";
```

```
    echo hiya igor;
    echo "'hiya igor' takes about $echo_timer_elapsed sec.";
}
```

```
location /echo_with_sleep {
    echo hello;
    echo_flush;  # ensure the client can see previous output immediately
    echo_sleep   2.5;  # in sec
    echo world;
}
```

```
# in the following example, accessing /echo yields
#   hello
#   world
#   blah
#   hiya
#   igor
location /echo {
    echo_before_body hello;
    echo_before_body world;
    proxy_pass $scheme://127.0.0.1:$server_port$request_uri/more;
    echo_after_body hiya;
    echo_after_body igor;
}
location /echo/more {
    echo blah;
}
```

```
# the output of /main might be
#   hello
#   world
#   took 0.000 sec for total.
# and the whole request would take about 2 sec to complete.
```

```
location /main {
    echo_reset_timer;

    # subrequests in parallel
    echo_location_async /sub1;
    echo_location_async /sub2;

    echo "took $echo_timer_elapsed sec for total.";
}
location /sub1 {
    echo_sleep 2;
    echo hello;
}
location /sub2 {
    echo_sleep 1;
    echo world;
}
```

```
# the output of /main might be
#   hello
#   world
#   took 3.003 sec for total.
# and the whole request would take about 3 sec to complete.
location /main {
    echo_reset_timer;

    # subrequests in series (chained by CPS)
    echo_location /sub1;
    echo_location /sub2;

    echo "took $echo_timer_elapsed sec for total.";
}
location /sub1 {
    echo_sleep 2;
    echo hello;
}
```

```
location /sub2 {
    echo_sleep 1;
    echo world;
}
```

```
# Accessing /dup gives
#   ------ END ------
location /dup {
  echo_duplicate 3 "--";
  echo_duplicate 1 " END ";
  echo_duplicate 3 "--";
  echo;
}
```

```
# /bighello will generate 1000,000,000 hello's.
location /bighello {
  echo_duplicate 1000_000_000 'hello';
}
```

```
# echo back the client request
location /echoback {
  echo_duplicate 1 $echo_client_request_headers;
  echo "\r";

  echo_read_request_body;

  echo_request_body;
}
```

```
# GET /multi will yields
#   querystring: foo=Foo
#   method: POST
```

```
#   body: hi
#   content length: 2
#   ///
#   querystring: bar=Bar
#   method: PUT
#   body: hello
#   content length: 5
#   ///
location /multi {
    echo_subrequest_async POST '/sub' -q 'foo=Foo' -b 'hi';
    echo_subrequest_async PUT '/sub' -q 'bar=Bar' -b 'hello';
}
location /sub {
    echo "querystring: $query_string";
    echo "method: $echo_request_method";
    echo "body: $echo_request_body";
    echo "content length: $http_content_length";
    echo '///';
}
```

```
# GET /merge?/foo.js&/bar/blah.js&/yui/baz.js will merge the .js resources together
location /merge {
    default_type 'text/javascript';
    echo_foreach_split '&' $query_string;
        echo "/* JS File $echo_it */";
        echo_location_async $echo_it;
        echo;
    echo_end;
}
```

```
# accessing /if?val=abc yields the "hit" output
# while /if?val=bcd yields "miss":
location ^~ /if {
    set $res miss;
    if ($arg_val ~* '^a') {
```

```
            set $res hit;
            echo $res;
        }
        echo $res;
    }
```

Back to TOC

Description

This module wraps lots of Nginx internal APIs for streaming input and output, parallel/sequential subrequests, timers and sleeping, as well as various meta data accessing.

Basically it provides various utilities that help testing and debugging of other modules by trivially emulating different kinds of faked subrequest locations.

People will also find it useful in real-world applications that need to

1. serve static contents directly from memory (loading from the Nginx config file).
2. wrap the upstream response with custom header and footer (kinda like the addition module but with contents read directly from the config file and Nginx variables).
3. merge contents of various "Nginx locations" (i.e., subrequests) together in a single main request (using echo_location and its friends).

This is a special dual-role module that can *lazily* serve as a content handler or register itself as an output filter only upon demand. By default, this module does not do anything at all.

Technially, this module has also demonstrated the following techniques that might be helpful for module writers:

1. Issue parallel subreqeusts directly from content handler.
2. Issue chained subrequests directly from content handler, by passing continuation along the

subrequest chain.
3. Issue subrequests with all HTTP 1.1 methods and even an optional faked HTTP request body.
4. Interact with the Nginx event model directly from content handler using custom events and timers, and resume the content handler back if necessary.
5. Dual-role module that can (lazily) serve as a content handler or an output filter or both.
6. Nginx config file variable creation and interpolation.
7. Streaming output control using output_chain, flush and its friends.
8. Read client request body from the content handler, and returns back (asynchronously) to the content handler after completion.
9. Use Perl-based declarative test suite to drive the development of Nginx C modules.

Back to TOC

Content Handler Directives

Use of the following directives register this module to the current Nginx location as a content handler. If you want to use another module, like the standard proxy module, as the content handler, use the filter directives provided by this module.

All the content handler directives can be mixed together in a single Nginx location and they're supposed to run sequentially just as in the Bash scripting language.

Every content handler directive supports variable interpolation in its arguments (if any).

The MIME type set by the standard default_type directive is respected by this module, as in:

```
location /hello {
  default_type text/plain;
  echo hello;
}
```

Then on the client side:

```
$ curl -I 'http://localhost/echo'
HTTP/1.1 200 OK
Server: nginx/0.8.20
Date: Sat, 17 Oct 2009 03:40:19 GMT
Content-Type: text/plain
Connection: keep-alive
```

Since the v0.22 release, all of the directives are allowed in the rewrite module's if directive block, for instance:

```
location ^~ /if {
    set $res miss;
    if ($arg_val ~* '^a') {
        set $res hit;
        echo $res;
    }
    echo $res;
}
```

Back to TOC

echo

syntax: *echo [options] <string>...*

default: *no*

context: *location, location if*

phase: *content*

Sends arguments joined by spaces, along with a trailing newline, out to the client.

Note that the data might be buffered by Nginx's underlying buffer. To force the output data flushed immediately, use the echo_flush command just after `echo`, as in

```
    echo hello world;
    echo_flush;
```

When no argument is specified, *echo* emits the trailing newline alone, just like the *echo* command in shell.

Variables may appear in the arguments. An example is

```
    echo The current request uri is $request_uri;
```

where $request_uri is a variable exposed by the ngx_http_core_module.

This command can be used multiple times in a single location configuration, as in

```
location /echo {
    echo hello;
    echo world;
}
```

The output on the client side looks like this

```
$ curl 'http://localhost/echo'
hello
world
```

Special characters like newlines (`\n`) and tabs (`\t`) can be escaped using C-style escaping

sequences. But a notable exception is the dollar sign (`$`). As of Nginx 0.8.20, there's still no clean way to esacpe this characters. (A work-around might be to use a `$echo_dollor` variable that is always evaluated to the constant `$` character. This feature will possibly be introduced in a future version of this module.)

As of the echo v0.28 release, one can suppress the trailing newline character in the output by using the `-n` option, as in

```
location /echo {
    echo -n "hello, ";
    echo "world";
}
```

Accessing `/echo` gives

```
$ curl 'http://localhost/echo'
hello, world
```

Leading `-n` in variable values won't take effect and will be emitted literally, as in

```
location /echo {
    set $opt -n;
    echo $opt "hello,";
    echo "world";
}
```

This gives the following output

```
$ curl 'http://localhost/echo'
-n hello,
world
```

One can output leading `-n` literals and other options using the special `--` option like this

```
location /echo {
    echo -- -n is an option;
}
```

which yields

```
$ curl 'http://localhost/echo'
-n is an option
```

Back to TOC

echo_duplicate

syntax: *echo_duplicate <count> <string>*

default: *no*

context: *location, location if*

phase: *content*

Outputs duplication of a string indicated by the second argument, using the times specified in the first argument.

For instance,

```
location /dup {
    echo_duplicate 3 "abc";
}
```

will lead to an output of `"abcabcabc"`.

Underscores are allowed in the count number, just like in Perl. For example, to emit 1000,000,000 instances of `"hello, world"`:

```
location /many_hellos {
    echo_duplicate 1000_000_000 "hello, world";
}
```

The `count` argument could be zero, but not negative. The second `string` argument could be an empty string ("") likewise.

Unlike the echo directive, no trailing newline is appended to the result. So it's possible to "abuse" this directive as a no-trailing-newline version of echo by using "count" 1, as in

```
location /echo_art {
    echo_duplicate 2 '---';
    echo_duplicate 1 ' END ';  # we don't want a trailing newline here
    echo_duplicate 2 '---';
    echo;  # we want a trailing newline here...
}
```

You get

```
------ END ------
```

This directive was first introduced in version 0.11.

Back to TOC

echo_flush

syntax: *echo_flush*

default: *no*

context: *location, location if*

phase: *content*

Forces the data potentially buffered by underlying Nginx output filters to send immediately to the client side via socket.

Note that techically the command just emits a ngx_buf_t object with `flush` slot set to 1, so certain weird third-party output filter module could still block it before it reaches Nginx's (last) write filter.

This directive does not take any argument.

Consider the following example:

```
location /flush {
   echo hello;

   echo_flush;

   echo_sleep 1;
   echo world;
}
```

Then on the client side, using curl to access `/flush`, you'll see the "hello" line immediately, but only after 1 second, the last "world" line. Without calling `echo_flush` in the example above, you'll most likely see no output until 1 second is elapsed due to the internal buffering of Nginx.

This directive will fail to flush the output buffer in case of subrequests get involved. Consider the following example:

```
location /main {
    echo_location_async /sub;
    echo hello;
    echo_flush;
}
location /sub {
    echo_sleep 1;
}
```

Then the client won't see "hello" appear even if `echo_flush` has been executed before the subrequest to `/sub` has actually started executing. The outputs of `/main` that are sent *after* echo_location_async will be postponed and buffered firmly.

This does *not* apply to outputs sent before the subrequest initiated. For a modified version of the example given above:

```
location /main {
    echo hello;
    echo_flush;
    echo_location_async /sub;
}
location /sub {
    echo_sleep 1;
}
```

The client will immediately see "hello" before `/sub` enters sleeping.

See also echo, echo_sleep, and echo_location_async.

Back to TOC

echo_sleep

syntax: *echo_sleep <seconds>*

default: *no*

context: *location, location if*

phase: *content*

Sleeps for the time period specified by the argument, which is in seconds.

This operation is non-blocking on server side, so unlike the echo_blocking_sleep directive, it won't block the whole Nginx worker process.

The period might takes three digits after the decimal point and must be greater than 0.001.

An example is

```
location /echo_after_sleep {
    echo_sleep 1.234;
    echo resumed!;
}
```

Behind the scene, it sets up a per-request "sleep" ngx_event_t object, and adds a timer using that custom event to the Nginx event model and just waits for a timeout on that event. Because the "sleep" event is per-request, this directive can work in parallel subrequests.

Back to TOC

echo_blocking_sleep

syntax: *echo_blocking_sleep <seconds>*

default: *no*

context: *location, location if*

phase: *content*

This is a blocking version of the echo_sleep directive.

See the documentation of echo_sleep for more detail.

Behind the curtain, it calls the ngx_msleep macro provided by the Nginx core which maps to usleep on POSIX-compliant systems.

Note that this directive will block the current Nginx worker process completely while being executed, so never use it in production environment.

Back to TOC

echo_reset_timer

syntax: *echo_reset_timer*

default: *no*

context: *location, location if*

phase: *content*

Reset the timer begin time to *now*, i.e., the time when this command is executed during request.

The timer begin time is default to the starting time of the current request and can be overridden by this directive, potentially multiple times in a single location. For example:

```
location /timed_sleep {
    echo_sleep 0.03;
```

```
        echo "$echo_timer_elapsed sec elapsed.";

        echo_reset_timer;

        echo_sleep 0.02;
        echo "$echo_timer_elapsed sec elapsed.";
    }
```

The output on the client side might be

```
$ curl 'http://localhost/timed_sleep'
0.032 sec elapsed.
0.020 sec elapsed.
```

The actual figures you get on your side may vary a bit due to your system's current activities.

Invocation of this directive will force the underlying Nginx timer to get updated to the current system time (regardless the timer resolution specified elsewhere in the config file). Furthermore, references of the $echo_timer_elapsed variable will also trigger timer update forcibly.

See also echo_sleep and $echo_timer_elapsed.

Back to TOC

echo_read_request_body

syntax: *echo_read_request_body*

default: *no*

context: *location, location if*

phase: *content*

Explicitly reads request body so that the $request_body variable will always have non-empty values (unless the body is so big that it has been saved by Nginx to a local temporary file).

Note that this might not be the original client request body because the current request might be a subrequest with a "artificial" body specified by its parent.

This directive does not generate any output itself, just like echo_sleep.

Here's an example for echo'ing back the original HTTP client request (both headers and body are included):

```
location /echoback {
  echo_duplicate 1 $echo_client_request_headers;
  echo "\r";
  echo_read_request_body;
  echo $request_body;
}
```

The content of `/echoback` looks like this on my side (I was using Perl's LWP utility to access this location on the server):

```
$ (echo hello; echo world) | lwp-request -m POST 'http://localhost/echoback'
POST /echoback HTTP/1.1
TE: deflate,gzip;q=0.3
Connection: TE, close
Host: localhost
User-Agent: lwp-request/5.818 libwww-perl/5.820
Content-Length: 12
Content-Type: application/x-www-form-urlencoded

hello
world
```

Because `/echoback` is the main request, $request_body holds the original client request body.

Before Nginx 0.7.56, it makes no sense to use this directive because $request_body was first introduced in Nginx 0.7.58.

This directive itself was first introduced in the echo module's v0.14 release.

Back to TOC

echo_location_async

syntax: *echo_location_async <location> [<url_args>]*

default: *no*

context: *location, location if*

phase: *content*

Issue GET subrequest to the location specified (first argument) with optional url arguments specified in the second argument.

As of Nginx 0.8.20, the `location` argument does *not* support named location, due to a limitation in the `ngx_http_subrequest` function. The same is true for its brother, the echo_location directive.

A very simple example is

```
location /main {
    echo_location_async /sub;
    echo world;
}
location /sub {
    echo hello;
}
```

Accessing `/main` gets

```
hello
world
```

Calling multiple locations in parallel is also possible:

```
location /main {
    echo_reset_timer;
    echo_location_async /sub1;
    echo_location_async /sub2;
    echo "took $echo_timer_elapsed sec for total.";
}
location /sub1 {
    echo_sleep 2; # sleeps 2 sec
    echo hello;
}
location /sub2 {
    echo_sleep 1; # sleeps 1 sec
    echo world;
}
```

Accessing `/main` yields

```
$ time curl 'http://localhost/main'
hello
world
took 0.000 sec for total.

real  0m2.006s
user  0m0.000s
sys   0m0.004s
```

You can see that the main handler `/main` does *not* wait the subrequests `/sub1` and `/sub2` to

complete and quickly goes on, hence the "0.000 sec" timing result. The whole request, however takes approximately 2 sec in total to complete because `/sub1` and `/sub2` run in parallel (or "concurrently" to be more accurate).

If you use echo_blocking_sleep in the previous example instead, then you'll get the same output, but with 3 sec total response time, because "blocking sleep" blocks the whole Nginx worker process.

Locations can also take an optional querystring argument, for instance

```
location /main {
    echo_location_async /sub 'foo=Foo&bar=Bar';
}
location /sub {
    echo $arg_foo $arg_bar;
}
```

Accessing `/main` yields

```
$ curl 'http://localhost/main'
Foo Bar
```

Querystrings is *not* allowed to be concatenated onto the `location` argument with "?" directly, for example, `/sub?foo=Foo&bar=Bar` is an invalid location, and shouldn't be fed as the first argument to this directive.

Technically speaking, this directive is an example that Nginx content handler issues one or more subrequests directly. AFAIK, the fancyindex module also does such kind of things ;)

Nginx named locations like `@foo` is *not* supported here.

This directive is logically equivalent to the GET version of echo_subrequest_async. For example,

```
echo_location_async /foo 'bar=Bar';
```

is logically equivalent to

```
echo_subrequest_async GET /foo -q 'bar=Bar';
```

But calling this directive is slightly faster than calling echo_subrequest_async using `GET` because we don't have to parse the HTTP method names like `GET` and options like `-q`.

There is a known issue with this directive when disabling the standard standard SSI module. See Known Issues for more details.

This directive is first introduced in version 0.09 of this module and requires at least Nginx 0.7.46.

Back to TOC

echo_location

syntax: *echo_location <location> [<url_args>]*

default: *no*

context: *location, location if*

phase: *content*

Just like the echo_location_async directive, but `echo_location` issues subrequests *in series* rather than in parallel. That is, the content handler directives following this directive won't be executed until the subrequest issued by this directive completes.

The final response body is almost always equivalent to the case when echo_location_async is used

instead, only if timing variables is used in the outputs.

Consider the following example:

```
location /main {
    echo_reset_timer;
    echo_location /sub1;
    echo_location /sub2;
    echo "took $echo_timer_elapsed sec for total.";
}
location /sub1 {
    echo_sleep 2;
    echo hello;
}
location /sub2 {
    echo_sleep 1;
    echo world;
}
```

The location `/main` above will take for total 3 sec to complete (compared to 2 sec if echo_location_async is used instead here). Here's the result in action on my machine:

```
$ curl 'http://localhost/main'
hello
world
took 3.003 sec for total.

real  0m3.027s
user  0m0.020s
sys   0m0.004s
```

This directive is logically equivalent to the GET version of echo_subrequest. For example,

```
echo_location /foo 'bar=Bar';
```

is logically equivalent to

```
echo_subrequest GET /foo -q 'bar=Bar';
```

But calling this directive is slightly faster than calling echo_subrequest using `GET` because we don't have to parse the HTTP method names like `GET` and options like `-q`.

Behind the scene, it creates an `ngx_http_post_subrequest_t` object as a *continuation* and passes it into the `ngx_http_subrequest` function call. Nginx will later reopen this "continuation" in the subrequest's `ngx_http_finalize_request` function call. We resumes the execution of the parent-request's content handler and starts to run the next directive (command) if any.

Nginx named locations like `@foo` is *not* supported here.

This directive was first introduced in the release v0.12.

See also echo_location_async for more details about the meaning of the arguments.

Back to TOC

echo_subrequest_async

syntax: *echo_subrequest_async <HTTP_method> <location> [-q <url_args>] [-b <request_body>] [-f <request_body_path>]*

default: *no*

context: *location, location if*

phase: *content*

Initiate an asynchronous subrequest using HTTP method, an optional url arguments (or querystring) and an optional request body which can be defined as a string or as a path to a file which contains the body.

This directive is very much like a generalized version of the echo_location_async directive.

Here's a small example demonstrating its usage:

```
location /multi {
    # body defined as string
    echo_subrequest_async POST '/sub' -q 'foo=Foo' -b 'hi';
    # body defined as path to a file, relative to nginx prefix path if not absolute
    echo_subrequest_async PUT '/sub' -q 'bar=Bar' -f '/tmp/hello.txt';
}
location /sub {
    echo "querystring: $query_string";
    echo "method: $echo_request_method";
    echo "body: $echo_request_body";
    echo "content length: $http_content_length";
    echo '///';
}
```

Then on the client side:

```
$ echo -n hello > /tmp/hello.txt
$ curl 'http://localhost/multi'
querystring: foo=Foo
method: POST
body: hi
content length: 2
///
querystring: bar=Bar
```

```
  method: PUT
  body: hello
  content length: 5
  ///
```

Here's more funny example using the standard proxy module to handle the subrequest:

```
location /main {
    echo_subrequest_async POST /sub -b 'hello, world';
}
location /sub {
    proxy_pass $scheme://127.0.0.1:$server_port/proxied;
}
location /proxied {
    echo "method: $echo_request_method.";

    # we need to read body explicitly here...or $echo_request_body
    #   will evaluate to empty ("")
    echo_read_request_body;

    echo "body: $echo_request_body.";
}
```

Then on the client side, we can see that

```
  $ curl 'http://localhost/main'
  method: POST.
  body: hello, world.
```

Nginx named locations like `@foo` is *not* supported here.

This directive takes several options:

```
-q <url_args>        Specify the URL arguments (or URL querystring) for the subrequest.
```

```
-f <path>           Specify the path for the file whose content will be serve as the
                    subrequest's request body.

-b <data>           Specify the request body data
```

This directive was first introduced in the release v0.15.

The `-f` option to define a file path for the body was introduced in the release v0.35.

See also the echo_subrequest and echo_location_async directives.

There is a known issue with this directive when disabling the standard standard SSI module. See Known Issues for more details.

Back to TOC

echo_subrequest

syntax: *echo_subrequest <HTTP_method> <location> [-q <url_args>] [-b <request_body>] [-f <request_body_path>]*

default: *no*

context: *location, location if*

phase: *content*

This is the synchronous version of the echo_subrequest_async directive. And just like echo_location, it does not block the Nginx worker process (while echo_blocking_sleep does), rather, it uses continuation to pass control along the subrequest chain.

See echo_subrequest_async for more details.

Nginx named locations like `@foo` is *not* supported here.

This directive was first introduced in the release v0.15.

Back to TOC

echo_foreach_split

syntax: *echo_foreach_split <delimiter> <string>*

default: *no*

context: *location, location if*

phase: *content*

Split the second argument `string` using the delimiter specified in the first argument, and then iterate through the resulting items. For instance:

```
location /loop {
  echo_foreach_split ',' $arg_list;
    echo "item: $echo_it";
  echo_end;
}
```

Accessing /main yields

```
$ curl 'http://localhost/loop?list=cat,dog,mouse'
item: cat
item: dog
item: mouse
```

As seen in the previous example, this directive should always be accompanied by an echo_end directive.

Parallel `echo_foreach_split` loops are allowed, but nested ones are currently forbidden.

The `delimiter` argument could contain *multiple* arbitrary characters, like

```
# this outputs "cat\ndog\nmouse\n"
echo_foreach_split -- '-a-' 'cat-a-dog-a-mouse';
  echo $echo_it;
echo_end;
```

Logically speaking, this looping structure is just the `foreach` loop combined with a `split` function call in Perl (using the previous example):

```
 foreach (split ',', $arg_list) {
     print "item $_\n";
 }
```

People will also find it useful in merging multiple `.js` or `.css` resources into a whole. Here's an example:

```
location /merge {
    default_type 'text/javascript';

    echo_foreach_split '&' $query_string;
        echo "/* JS File $echo_it */";
        echo_location_async $echo_it;
        echo;
    echo_end;
}
```

Then accessing /merge to merge the `.js` resources specified in the query string:

```
$ curl 'http://localhost/merge?/foo/bar.js&/yui/blah.js&/baz.js'
```

One can also use third-party Nginx cache module to cache the merged response generated by the `/merge` location in the previous example.

This directive was first introduced in the release v0.17.

Back to TOC

echo_end

syntax: *echo_end*

default: *no*

context: *location, location if*

phase: *content*

This directive is used to terminate the body of looping and conditional control structures like echo_foreach_split.

This directive was first introduced in the release v0.17.

Back to TOC

echo_request_body

syntax: *echo_request_body*

default: *no*

context: *location, location if*

phase: *content*

Outputs the contents of the request body previous read.

Behind the scene, it's implemented roughly like this:

```
if (r->request_body && r->request_body->bufs) {
    return ngx_http_output_filter(r, r->request_body->bufs);
}
```

Unlike the $echo_request_body and $request_body variables, this directive will show the whole request body even if some parts or all parts of it are saved in temporary files on the disk.

It is a "no-op" if no request body has been read yet.

This directive was first introduced in the release v0.18.

See also echo_read_request_body and the chunkin module.

Back to TOC

echo_exec

syntax: *echo_exec <location> [<query_string>]*

syntax: *echo_exec <named_location>*

default: *no*

context: *location, location if*

phase: *content*

Does an internal redirect to the location specified. An optional query string can be specified for normal locations, as in

```
location /foo {
    echo_exec /bar weight=5;
}
location /bar {
    echo $arg_weight;
}
```

Or equivalently

```
location /foo {
    echo_exec /bar?weight=5;
}
location /bar {
    echo $arg_weight;
}
```

Named locations are also supported. Here's an example:

```
location /foo {
    echo_exec @bar;
}
location @bar {
    # you'll get /foo rather than @bar
    #  due to a potential bug in nginx.
    echo $echo_request_uri;
}
```

But query string (if any) will always be ignored for named location redirects due to a limitation in the `ngx_http_named_location` function.

Never try to echo things before the `echo_exec` directive or you won't see the proper response of the location you want to redirect to. Because any echoing will cause the original location handler to send HTTP headers before the redirection happens.

Technically speaking, this directive exposes the Nginx internal API functions `ngx_http_internal_redirect` and `ngx_http_named_location`.

This directive was first introduced in the v0.21 release.

Back to TOC

echo_status

syntax: *echo_status <status-num>*

default: *echo_status 200*

context: *location, location if*

phase: *content*

Specify the default response status code. Default to `200`. This directive is declarative and the relative order with other echo-like directives is not important.

Here is an example,

```
location = /bad {
    echo_status 404;
    echo "Something is missing...";
```

```
}
```

then we get a response like this:

```
HTTP/1.1 404 Not Found
Server: nginx/1.2.1
Date: Sun, 24 Jun 2012 03:58:18 GMT
Content-Type: text/plain
Transfer-Encoding: chunked
Connection: keep-alive

Something is missing...
```

This directive was first introduced in the `v0.40` release.

Back to TOC

Filter Directives

Use of the following directives trigger the filter registration of this module. By default, no filter will be registered by this module.

Every filter directive supports variable interpolation in its arguments (if any).

Back to TOC

echo_before_body

syntax: *echo_before_body [options] [argument]...*

default: *no*

context: *location, location if*

phase: *output filter*

It's the filter version of the echo directive, and prepends its output to the beginning of the original outputs generated by the underlying content handler.

An example is

```
location /echo {
    echo_before_body hello;
    proxy_pass $scheme://127.0.0.1:$server_port$request_uri/more;
}
location /echo/more {
    echo world
}
```

Accessing `/echo` from the client side yields

```
  hello
  world
```

In the previous sample, we borrow the standard proxy module to serve as the underlying content handler that generates the "main contents".

Multiple instances of this filter directive are also allowed, as in:

```
location /echo {
    echo_before_body hello;
    echo_before_body world;
    echo !;
}
```

On the client side, the output is like

```
$ curl 'http://localhost/echo'
hello
world
!
```

In this example, we also use the content handler directives provided by this module as the underlying content handler.

This directive also supports the `-n` and `--` options like the echo directive.

This directive can be mixed with its brother directive echo_after_body.

Back to TOC

echo_after_body

syntax: *echo_after_body [argument]...*

default: *no*

context: *location, location if*

phase: *output filter*

It's very much like the echo_before_body directive, but *appends* its output to the end of the original outputs generated by the underlying content handler.

Here's a simple example:

```
location /echo {
```

```
    echo_after_body hello;
    proxy_pass http://127.0.0.1:$server_port$request_uri/more;
}
location /echo/more {
    echo world
}
```

Accessing `/echo` from the client side yields

```
world
hello
```

Multiple instances are allowed, as in:

```
location /echo {
    echo_after_body hello;
    echo_after_body world;
    echo i;
    echo say;
}
```

The output on the client side while accessing the `/echo` location looks like

```
i
say
hello
world
```

This directive also supports the `-n` and `--` options like the echo directive.

This directive can be mixed with its brother directive echo_before_body.

Back to TOC

Variables

Back to TOC

$echo_it

This is a "topic variable" used by echo_foreach_split, just like the `$_` variable in Perl.

Back to TOC

$echo_timer_elapsed

This variable holds the seconds elapsed since the start of the current request (might be a subrequest though) or the last invocation of the echo_reset_timer command.

The timing result takes three digits after the decimal point.

References of this variable will force the underlying Nginx timer to update to the current system time, regardless the timer resolution settings elsewhere in the config file, just like the echo_reset_timer directive.

Back to TOC

$echo_request_body

Evaluates to the current (sub)request's request body previously read if no part of the body has been saved to a temporary file. To always show the request body even if it's very large, use the echo_request_body directive.

Back to TOC

$echo_request_method

Evaluates to the HTTP request method of the current request (it can be a subrequest).

Behind the scene, it just takes the string data stored in `r->method_name`.

Compare it to the $echo_client_request_method variable.

At least for Nginx 0.8.20 and older, the $request_method variable provided by the http core module is actually doing what our $echo_client_request_method is doing.

This variable was first introduced in our v0.15 release.

Back to TOC

$echo_client_request_method

Always evaluates to the main request's HTTP method even if the current request is a subrequest.

Behind the scene, it just takes the string data stored in `r->main->method_name`.

Compare it to the $echo_request_method variable.

This variable was first introduced in our v0.15 release.

Back to TOC

$echo_client_request_headers

Evaluates to the original client request's headers.

Just as the name suggests, it will always take the main request (or the client request) even if it's currently executed in a subrequest.

A simple example is below:

```
location /echoback {
   echo "headers are:"
   echo $echo_client_request_headers;
}
```

Accessing `/echoback` yields

```
$ curl 'http://localhost/echoback'
headers are
GET /echoback HTTP/1.1
User-Agent: curl/7.18.2 (i486-pc-linux-gnu) libcurl/7.18.2 OpenSSL/0.9.8g
Host: localhost:1984
Accept: */*
```

Behind the scene, it recovers `r->main->header_in` (or the large header buffers, if any) on the C level and does not construct the headers itself by traversing parsed results in the request object.

This variable was first introduced in version 0.15.

Back to TOC

$echo_cacheable_request_uri

Evaluates to the parsed form of the URI (usually led by `/`) of the current (sub-)request. Unlike the $echo_request_uri variable, it is cacheable.

See $echo_request_uri for more details.

This variable was first introduced in version 0.17.

Back to TOC

$echo_request_uri

Evaluates to the parsed form of the URI (usually led by `/`) of the current (sub-)request. Unlike the $echo_cacheable_request_uri variable, it is *not* cacheable.

This is quite different from the $request_uri variable exported by the ngx_http_core_module, because `$request_uri` is the *unparsed* form of the current request's URI.

This variable was first introduced in version 0.17.

Back to TOC

$echo_incr

It is a counter that always generate the current counting number, starting from 1. The counter is always associated with the main request even if it is accessed within a subrequest.

Consider the following example

```
location /main {
    echo "main pre: $echo_incr";
    echo_location_async /sub;
    echo_location_async /sub;
    echo "main post: $echo_incr";
}
location /sub {
    echo "sub: $echo_incr";
}
```

Accessing `/main` yields

```
main pre: 1
sub: 3
sub: 4
main post: 2
```

This directive was first introduced in the v0.18 release.

Back to TOC

$echo_response_status

Evaluates to the status code of the current (sub)request, null if not any.

Behind the scene, it's just the textual representation of `r->headers_out->status`.

This directive was first introduced in the v0.23 release.

Back to TOC

Installation

You're recommended to install this module (as well as the Nginx core and many other goodies) via the ngx_openresty bundle. See the detailed instructions for downloading and installing ngx_openresty into your system. This is the easiest and most safe way to set things up.

Alternatively, you can install this module manually with the Nginx source:

Grab the nginx source code from nginx.org, for example, the version 1.7.7 (see nginx compatibility), and then build the source with this module:

```
$ wget 'http://nginx.org/download/nginx-1.7.7.tar.gz'
$ tar -xzvf nginx-1.7.7.tar.gz
$ cd nginx-1.7.7/

# Here we assume you would install you nginx under /opt/nginx/.
$ ./configure --prefix=/opt/nginx \
    --add-module=/path/to/echo-nginx-module

$ make -j2
$ make install
```

Download the latest version of the release tarball of this module from echo-nginx-module file list.

Also, this module is included and enabled by default in the ngx_openresty bundle.

Back to TOC

Compatibility

The following versions of Nginx should work with this module:

- **1.7.x** (last tested: 1.7.7)
- **1.6.x**
- **1.5.x** (last tested: 1.5.12)
- **1.4.x** (last tested: 1.4.4)
- **1.3.x** (last tested: 1.3.7)
- **1.2.x** (last tested: 1.2.9)
- **1.1.x** (last tested: 1.1.5)
- **1.0.x** (last tested: 1.0.11)
- **0.9.x** (last tested: 0.9.4)
- **0.8.x** (last tested: 0.8.54)

- **0.7.x >= 0.7.21** (last tested: 0.7.68)

In particular,

- the directive echo_location_async and its brother echo_subrequest_async do *not* work with **0.7.x < 0.7.46**.
- the echo_after_body directive does *not* work at all with nginx **< 0.8.7**.
- the echo_sleep directive cannot be used after echo_location or echo_subrequest for nginx **< 0.8.11**.

Earlier versions of Nginx like 0.6.x and 0.5.x will *not* work at all.

If you find that any particular version of Nginx above 0.7.21 does not work with this module, please consider reporting a bug.

Back to TOC

Known Issues

Due to an unknown bug in Nginx (it still exists in Nginx 1.7.7), the standard SSI module is required to ensure that the contents of the subrequests issued by echo_locatoin_async and echo_subrequest_async are correctly merged into the output chains of the main one. Fortunately, the SSI module is enabled by default during Nginx's `configure` process.

If calling this directive without SSI module enabled, you'll get truncated response without contents of any subrequests and get an alert message in your Nginx's `error.log`, like this:

```
[alert] 24212#0: *1 the http output chain is empty, client: 127.0.0.1, ...
```

Back to TOC

Modules that use this module for testing

The following modules take advantage of this `echo` module in their test suite:

- The memc module that supports almost the whole memcached TCP protocol.
- The chunkin module that adds HTTP 1.1 chunked input support to Nginx.
- The headers_more module that allows you to add, set, and clear input and output headers under the conditions that you specify.
- The `echo` module itself.

Please mail me other modules that use `echo` in any form and I'll add them to the list above :)

Back to TOC

Community

Back to TOC

English Mailing List

The openresty-en mailing list is for English speakers.

Back to TOC

Chinese Mailing List

The openresty mailing list is for Chinese speakers.

Back to TOC

Report Bugs

Although a lot of effort has been put into testing and code tuning, there must be some serious bugs lurking somewhere in this module. So whenever you are bitten by any quirks, please don't hesitate to

1. create a ticket on the issue tracking interface provided by GitHub,
2. or send a bug report, questions, or even patches to the OpenResty Community.

Back to TOC

Source Repository

Available on github at agentzh/echo-nginx-module.

Back to TOC

Changes

The changes of every release of this module can be obtained from the ngx_openresty bundle's change logs:

http://openresty.org/#Changes

Back to TOC

Test Suite

This module comes with a Perl-driven test suite. The test cases are declarative too. Thanks to the Test::Nginx module in the Perl world.

To run it on your side:

```
$ PATH=/path/to/your/nginx-with-echo-module:$PATH prove -r t
```

You need to terminate any Nginx processes before running the test suite if you have changed the Nginx server binary.

Because a single nginx server (by default, `localhost:1984`) is used across all the test scripts (`.t` files), it's meaningless to run the test suite in parallel by specifying `-jN` when invoking the `prove` utility.

Some parts of the test suite requires standard modules proxy, rewrite and SSI to be enabled as well when building Nginx.

Back to TOC

TODO

- Fix the echo_after_body directive in subrequests.
- Add directives *echo_read_client_request_body* and *echo_request_headers*.
- Add new directive *echo_log* to use Nginx's logging facility directly from the config file and specific loglevel can be specified, as in

```
echo_log debug "I am being called.";
```

- Add support for options `-h` and `-t` to echo_subrequest_async and echo_subrequest. For

example

```
echo_subrequest POST /sub -q 'foo=Foo&bar=Bar' -b 'hello' -t 'text/plan' -h 'X-My-Header: blah
```

- Add options to control whether a subrequest should inherit cached variables from its parent request (i.e. the current request that is calling the subrequest in question). Currently none of the subrequests issued by this module inherit the cached variables from the parent request.
- Add new variable *$echo_active_subrequests* to show `r->main->count - 1`.
- Add the *echo_file* and *echo_cached_file* directives.
- Add new varaible *$echo_request_headers* to accompany the existing $echo_client_request_headers variable.
- Add new directive *echo_foreach*, as in

```
echo_foreach 'cat' 'dog' 'mouse';
  echo_location_async "/animals/$echo_it";
echo_end;
```

- Add new directive *echo_foreach_range*, as in

```
echo_foreach_range '[1..100]' '[a-zA-z0-9]';
  echo_location_async "/item/$echo_it";
echo_end;
```

- Add new directive *echo_repeat*, as in

```
echo_repeat 10 $i {
    echo "Page $i";
    echo_location "/path/to/page/$i";
}
```

This is just another way of saying

```
echo_foreach_range $i [1..10];
    echo "Page $i";
    echo_location "/path/to/page/$i";
echo_end;
```

Thanks Marcus Clyne for providing this idea.

- Add new variable $echo_random which always returns a random non-negative integer with the lower/upper limit specified by the new directives `echo_random_min` and `echo_random_max`. For example,

```
echo_random_min  10
echo_random_max  200
echo "random number: $echo_random";
```

Thanks Marcus Clyne for providing this idea.

Back to TOC

Getting involved

You'll be very welcomed to submit patches to the author or just ask for a commit bit to the source repository on GitHub.

Back to TOC

Author

Yichun "agentzh" Zhang (章亦春) <*agentzh@gmail.com*>, CloudFlare Inc.

This wiki page is also maintained by the author himself, and everybody is encouraged to improve this page as well.

Back to TOC

Copyright & License

THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Back to TOC

See Also

- The original blog post about this module's initial development.
- The standard addition filter module.
- The standard proxy module.
- The ngx_openresty bundle.