



This repository Search

Pull requests Issues Gist



openresty / echo-nginx-module

Watch

68

Star

462

Fork

136

&lt;&gt; Code

Issues 8

Pull requests 0

Projects 0

Wiki

Pulse

Graphs

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

464 commits

2 branches

79 releases

7 contributors

BSD-2-Clause

Branch: master

New pull request

Create new file

Upload files

Find file

Clone or download



agentzh doc: updated copyright notice.

Latest commit cafc87b on Dec 18, 2016

src	bugfix: fixed compilation failures when specifying the C compiler opt...	8 months ago
t	a follow-up fix in the previous commit.	a year ago
util	bugfix: fixed crashes in \$echo_client_request_headers for HTTP/2 requ...	8 months ago
.gitattributes	added a .gitattributes file to correct GitHub's language tag.	a year ago
.gitignore	updated .gitignore a bit.	4 years ago
.travis.yml	travis-ci: tests against nginx 1.10.0 instead of 1.8.1.	9 months ago
LICENSE	updated the LICENSE file to match README.	2 years ago
README.markdown	doc: updated copyright notice.	2 months ago
config	bugfix: config: avoids duplicate inclusion of the http_postpone module.	10 months ago

 [valgrind.suppress](#)

suppressed a valgrind false positive in libdl.

3 years 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

---

- [Name](#)
- [Status](#)
- [Version](#)
- [Synopsis](#)
- [Description](#)
- [Content Handler Directives](#)
  - [echo](#)
  - [echo\\_duplicate](#)
  - [echo\\_flush](#)
  - [echo\\_sleep](#)
  - [echo\\_blocking\\_sleep](#)
  - [echo\\_reset\\_timer](#)

- `echo_read_request_body`
- `echo_location_async`
- `echo_location`
- `echo_subrequest_async`
- `echo_subrequest`
- `echo_foreach_split`
- `echo_end`
- `echo_request_body`
- `echo_exec`
- `echo_status`
- **Filter Directives**
  - `echo_before_body`
  - `echo_after_body`
- **Variables**
  - `$echo_it`
  - `$echo_timer_elapsed`
  - `$echo_request_body`
  - `$echo_request_method`
  - `$echo_client_request_method`
  - `$echo_client_request_headers`
  - `$echo_cacheable_request_uri`
  - `$echo_request_uri`
  - `$echo_incr`
  - `$echo_response_status`
- **Installation**

- [Compatibility](#)
- [Modules that use this module for testing](#)
- [Community](#)
  - [English Mailing List](#)
  - [Chinese Mailing List](#)
- [Report Bugs](#)
- [Source Repository](#)
- [Changes](#)
- [Test Suite](#)
- [TODO](#)
- [Getting involved](#)
- [Author](#)
- [Copyright & License](#)
- [See Also](#)

## Status

---

This module is production ready.

## Version

---

This document describes ngx\_echo [v0.60](#) released on 15 August 2016.

## 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 subrequeusts 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 ( `$` ). As of Nginx 0.8.20, there's still no clean way to escape this character. (A work-around might be to use a `$echo_dollar` 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
```

Use this form when you want to output anything leading with a dash ( `-` ).

[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 count specified in the first argument.

For instance,

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

will lead to the 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 -----
```

But use of the `-n` option in [echo](#) is more appropriate for this purpose.

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 technically 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`.

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.

[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 is always evaluated to an empty value in HTTP/2 requests for now due to the current implementation.

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 [OpenResty bundle](#). See [the detailed instructions](#) for downloading and installing 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 1.11.2 (see [nginx compatibility](#)), and then build the source with this module:

```
$ wget 'http://nginx.org/download/nginx-1.11.2.tar.gz'
$ tar -xzf nginx-1.11.2.tar.gz
$ cd nginx-1.11.2/

# 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](#).



Starting from NGINX 1.9.11, you can also compile this module as a dynamic module, by using the `--add-dynamic-module=PATH` option instead of `--add-module=PATH` on the `./configure` command line above. And then you can explicitly load the module in your `nginx.conf` via the [load\\_module](#) directive, for example,

```
load_module /path/to/modules/ngx_http_echo_module.so;
```

Also, this module is included and enabled by default in the [OpenResty bundle](#).

[Back to TOC](#)

## Compatibility

---

The following versions of Nginx should work with this module:

- **1.11.x** (last tested: 1.11.2)
- **1.10.x**
- **1.9.x** (last tested: 1.9.15)
- **1.8.x**
- **1.7.x** (last tested: 1.7.10)
- **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](#)

## 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 [openresty/echo-nginx-module](https://github.com/openresty/echo-nginx-module).

[Back to TOC](#)

# Changes

---

The changes of every release of this module can be obtained from the 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/plain' -h 'X-My-Header: blah 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 `$echo_active_subrequests` to show `r->main->count - 1`.
- Add the [echo\\_file](#) and [echo\\_cached\\_file](#) directives.
- Add new variable `$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](mailto:agentzh@gmail.com)>, OpenResty 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

---

Copyright (c) 2009-2016, Yichun "agentzh" Zhang (章亦春) [agentzh@gmail.com](mailto:agentzh@gmail.com), OpenResty Inc.

This module is licensed under the terms of the BSD license.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following

conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF 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 [OpenResty](#) bundle.

[Back to TOC](#)



