Web Server, LLC, OGRN 1227700436578, INN 9704151517
ul. Vyatskaya, d. 27, str. 7, Moscow, Russia, 127015

User Guide

version 1.9.1

Web Server, LLC

May 29, 2025

Contents

CHAPTER 1

Annotation

This document contains the information necessary for the operation of the Angie PRO software.

CHAPTER 2

General Information

Angie PRO is the only commercial web server developed and localized in Russia.

A web server is a class of software that provides access to network resources via the HTTP protocol to end users. Angie PRO, for example, can be used to operate websites, mobile applications, self-service kiosks in the subway, and multimedia systems on long-distance trains. Every time a user opens a website, uses a mobile application, interacts with a self-service kiosk in the subway, or even with a multimedia system on the "Sapsan" train, the user's request can be processed by Angie PRO.

Angie PRO is:

- **A general-purpose web server.** Written in C.
- **An L4-L7 load balancer.** Allows load balancing between servers for both TCP/UDP protocols and HTTP.
- **A proxy and caching server.** Enables faster operation of web services through a flexible caching mechanism.
- **Available on all popular platforms.** Compiled and tested on Alpine, Debian, Oracle, RED OS, Rocky, and Ubuntu.
- **High performance.** One of the most efficient web servers in the world.

Why choose Angie PRO:

- **Compatibility with NGINX OSS.** Angie PRO is fully compatible with Nginx, allowing any existing Nginx user to transition to Angie PRO without significant costs or service downtime.
- **Enhanced statistics and real-time monitoring.** Angie PRO offers complete real-time server load monitoring, enabling dynamic configuration management based on load profiles and ensuring full service availability.
- **Dynamic configuration of proxied server groups.** Allows management of proxied server group settings through a convenient REST interface without service interruption.
- **Cache element removal.** Provides the ability to remove cache elements via a user-friendly API without service downtime.
- **Active health checks for proxied servers.** Checks for "liveness" and proxies only to those groups of proxied servers that respond according to a specified algorithm.
- **Dynamic key-value storage.** Enables dynamic management of Angie PRO configuration variables via HTTP API.

- **Dynamic DNS updates.**
- **Session-affinity proxying.**
- **Repository with dynamic third-party modules.** Angie PRO supports most NGINX third-party modules and allows for seamless installation, guaranteeing functionality and support.
- **Shared memory zone synchronization.** Capability to use cache zones, limit_req, etc., in the Angie PRO cluster.
- **Hiding or personal branding of the server name in response headers.** Ability to change or hide the name and version of the web server from users.

A list of foreign software with similar functional characteristics to Angie PRO includes Nginx, Nginx Plus, Apache, Envoy, products utilizing NGINX solutions (OpenResty, Tengine, Cloudflare), and Yandex's cloud solutions.

CHAPTER 3

Configuration

This page contains articles, guides, tips, and instructions on configuring Angie.

3.1 General

These articles cover installation and configuration of Angie, starting and stopping the web server, managing it, as well as various aspects of request handling and interaction with other servers.

3.1.1 Configuration Files

Angie uses a text-based configuration file. By default, this file is named `angie.conf` and is located according to the --conf-path build parameter, typically in the `/etc/angie` directory.

A configuration file generally consists of the following contexts:

- `events` – General connection processing
- `http` – HTTP traffic
- `mail` – Mail traffic
- `stream` – TCP and UDP traffic
- `wasm_modules` – WASM runtime

Directives that are placed outside of these contexts are considered to be in the `main` context:

```
user angie; # a directive in the 'main' context

events {

    # configuration of connection processing
}

http {

    # Configuration specific to HTTP and affecting all virtual servers

    server {
```

```
        # configuration of HTTP virtual server 1
        location /one {

            # configuration for processing URIs starting with '/one'
        }
        location /two {

            # configuration for processing URIs starting with '/two'
        }
    }

    server {

        # configuration of HTTP virtual server 2
    }
}

stream {

    # Configuration specific to TCP/UDP and affecting all virtual servers
    server {

        # configuration of TCP virtual server 1
    }
}
```

To simplify configuration management, we recommend using the *include* directive in the main `angie.conf` file to reference the contents of feature-specific files:

```
include /etc/angie/http.d/*.conf;
include /etc/angie/stream.d/*.conf;
```

Inheritance

In general, a child context (one that is contained within another context, which is considered its parent) inherits the settings of directives defined at the parent level. Some directives can appear in multiple contexts; in such cases, you can override the settings inherited from the parent by including the directive in the child context.

Syntax

Measurement Units

You can specify sizes using the following units:

No suffix	Bytes
`k`, `K`	Kilobytes
`m`, `M`	Megabytes
`g`, `G`	Gigabytes

For example: `1024`, `8k`, `1m`, `16g`.

Time intervals can be specified in milliseconds, seconds, minutes, hours, days, and so on, using the following suffixes:

`ms`	Milliseconds
`s`	Seconds
`m`	Minutes
`h`	Hours
`d`	Days
`w`	Weeks
`M`	Months (assumed equal to 30 days)
`y`	Years (assumed equal to 365 days)

Multiple units can be combined in a single value by specifying them in order from the most significant to the least significant, optionally separated by whitespace. For example, `"1h 30m"` specifies the same duration as `"90m"` or `"5400s"`. A value without a suffix is interpreted as seconds. It is recommended to always specify a suffix.

Some time intervals can only be specified with second-level resolution.

Directives

Each directive consists of a name and a set of parameters. If any part of a directive needs to contain spaces, it should be enclosed in quotes or escape the spaces:

```
add_header X-MyHeader "foo bar";
add_header X-MyHeader foo\ bar;
```

If a named parameter needs spaces and you use quotes, its name must be enclosed in quotes as well:

```
server example.com "sid=server 1";
```

Setting up Hashes

To efficiently process static sets of data, such as server names, the *map* directive values, MIME types, and request header names, Angie utilizes hash tables. During startup and each reconfiguration, Angie determines the optimal size for these hash tables to ensure that the bucket size, which stores keys with identical hash values, does not exceed the configured parameter (*hash bucket size*). The table size is measured in buckets and is adjusted until it exceeds the *hash max size* parameter. Most hash tables have corresponding directives to adjust these parameters, such as *server_names_hash_max_size* and *server_names_hash_bucket_size* for server names.

The *hash bucket size* parameter is aligned to a multiple of the processor's cache line size. This alignment enhances key search efficiency on modern processors by reducing the number of memory accesses. If the *hash bucket size* is equal to one cache line size, the maximum number of memory accesses during a key search will be two: one to compute the bucket address and another to search inside the bucket. Therefore, if Angie indicates that either the *hash max size* or *hash bucket size* should be increased, start by increasing the *hash max size*.

Reloading Configuration

To apply changes to the configuration file, it must be reloaded. You can either restart the Angie process with a configuration syntax check beforehand:

```
$ sudo angie -t && sudo service angie restart
```

Alternatively, you can reload the service to apply the new configuration without interrupting the processing of current requests:

```
$ sudo angie -t && sudo service angie reload
```

3.1.2 Run-Time Control

To start Angie, use `systemd` with the following command:

```
$ sudo service angie start
```

It is recommended to check the configuration syntax beforehand. Here is how:

```
$ sudo angie -t && sudo service angie start
```

To reload the configuration:

```
$ sudo angie -t && sudo service angie reload
```

To stop Angie:

```
$ sudo service angie stop
```

After installation, run the following command to ensure that Angie is up and running:

```
$ curl localhost:80
```

> **Note**
>
> The methods for running the open-source version of Angie may vary depending on the installation method.

Angie has one master process and several worker processes. The master process is responsible for reading and evaluating the configuration and maintaining the worker processes. Worker processes handle the actual request processing. Angie uses an event-based model and OS-dependent mechanisms to efficiently distribute requests among the worker processes. The number of worker processes is defined in the configuration file and may be either fixed for a given configuration or automatically adjusted based on the number of available CPU cores (see *worker_processes*).

When configured, Angie will also flush certain shared memory zones (currently, the `keys_zone` in *proxy_cache_path*) to the disk before exiting, so a newly started master process can restore them with improved performance. If the restore fails due to a change in zone size, binary version incompatibility, or other reasons, Angie will log an alert (`failed to restore zone at address`) and will not use the zone restore mechanism.

Using Signals

Angie can also be controlled using signals. By default, the process ID of the master process is written to the file `/run/angie.pid`. This filename can be changed at configuration time or in `angie.conf` using the *pid* directive. The master process supports the following signals:

`TERM`, `INT`	Fast shutdown
`QUIT`	*Graceful* shutdown
`HUP`	Reload configuration, update time zone (only for FreeBSD and Linux), start new worker processes with the updated configuration, *gracefully* shut down old worker processes
`USR1`	Reopen log files
`USR2`	Upgrade the executable file
`WINCH`	*Graceful* shutdown of worker processes

You can send signals using `kill`:

```
$ sudo kill -QUIT $(cat /run/angie.pid)
```

Individual worker processes can also be controlled using signals, although this is optional. The supported signals are:

TERM, INT	Fast shutdown
QUIT	*Graceful* shutdown
USR1	Reopen log files
WINCH	Abnormal termination for debugging (requires *debug_points* to be enabled)

Changing Configuration

In order for Angie to re-read the configuration file, a HUP signal should be sent to the master process. The master process first checks the syntax validity and then attempts to apply the new configuration, which includes opening new log files and listen sockets. If applying the new configuration fails, the master process rolls back the changes and continues operating with the old configuration. If the application succeeds, the master process starts new worker processes and sends messages to the old worker processes, requesting them to shut down *gracefully*. The old worker processes close their listen sockets and continue to service existing clients. After all clients have been served, the old worker processes are shut down.

Angie tracks configuration changes for each process. Generation numbers start at 1 when the server is first started. These numbers are incremented with each configuration reload and are visible in the process titles:

```
$ sudo angie
$ ps aux | grep angie

   angie: master process v1.9.1 #1 [angie]
   angie: worker process #1
```

After a successful configuration reload (regardless of whether there are actual changes), Angie increments the generation number for processes that received the new configuration:

```
$ sudo kill -HUP $(cat /run/angie.pid)
$ ps aux | grep angie

   angie: master process v1.9.1 #2 [angie]
   angie: worker process #2
```

If any worker processes from previous generations continue to operate, they will become immediately visible:

```
$ ps aux | grep angie

   angie: worker process #1
   angie: worker process #2
```

Note

Do not confuse the configuration generation number with a 'process number'; Angie does not use continuous process numbering for practical purposes.

Rotating Log Files

To rotate log files, first rename the files. Then, send a `USR1` signal to the master process. The master process will re-open all currently open log files and assign them to an unprivileged user under which the worker processes are running. After successfully re-opening the files, the master process closes all open files and notifies the worker processes to re-open their log files. Worker processes will also open the new files and close the old ones immediately. As a result, the old files become available for post-processing, such as compression, almost immediately.

On-the-fly Executable Upgrade

To upgrade the server executable, first replace the old executable file with the new one. Then, send a `USR2` signal to the master process. The master process will rename its current file with the process ID to a new file with the `.oldbin` suffix, e.g., `/usr/local/angie/logs/angie.pid.oldbin`, and then start the new executable, which in turn starts new worker processes.

Note that the old master process does not close its listen sockets and can be managed to restart its worker processes if necessary. If the new executable does not perform as expected, you can take one of the following actions:

- Send the `HUP` signal to the old master process. This will start new worker processes without re-reading the configuration. You can then shut down all new processes *gracefully* by sending the `QUIT` signal to the new master process.
- Send the `TERM` signal to the new master process. It will send a message to its worker processes, requesting them to exit immediately. If any processes do not exit, send the `KILL` signal to force them to exit. When the new master process exits, the old master process will automatically start new worker processes.

If the new master process exits, the old master process will remove the `.oldbin` suffix from the file name with the process ID.

If the upgrade is successful, send the `QUIT` signal to the old master process, and only the new processes will remain.

When configured, Angie will also flush certain shared memory zones (currently, the `keys_zone` in *proxy_cache_path*) to the disk before upgrading, so a newly started master process can restore them with improved performance. If the restore fails due to a change in zone size, binary version incompatibility, or other reasons, Angie will log an alert (`failed to restore zone at address`) and will not use the zone restore mechanism.

Command-Line Options

`-h`, `-?`	Display help for command-line parameters, then exit.
`--build-env`	Display auxiliary information about the build environment, then exit.
`-c` *file*	Use *file* as the configuration file instead of the *default file*.
`-e` *file*	Use *file* as the error log file instead of the *default file*. The special value `stderr` specifies the standard error output.
`-g` *directives*	Apply additional *global configuration directives*, for example: `angie -g "pid /var/run/angie.pid; worker_processes `sysctl -n hw.ncpu`;"`.
`-m`, `-M`	Display a list of built-in (`-m`) or built-in and loaded (`-M`) modules, then exit.
`-p` *prefix*	Specify the *prefix* path for `angie` (the directory where server files are located; the default is `/usr/local/angie/`).
`-q`	Only display error messages if `-t` or `-T` is set; otherwise, do nothing.
`-s` *signal*	Send a *signal* to the master process, such as `stop`, `quit`, `reopen`, `reload`, and so on.
`-t`	Test the configuration file, then exit. Angie checks the configuration syntax and recursively includes any files mentioned in it.
`-T`	Same as `-t`, but also outputs the summary configuration to standard output after recursively including all files mentioned in the configuration.
`-v`	Display the Angie version, then exit.
`-V`	Display the Angie version, the compiler version, build time and the build parameters used, then exit.

3.1.3 Connections, Sessions, Requests, Logs

Connection processing mechanisms

Angie supports various connection processing methods. The availability of a specific method depends on the platform being used. On platforms that support multiple methods, Angie typically selects the most efficient method automatically. However, if necessary, a connection processing method can be explicitly chosen using the *use* directive.

The following connection processing methods are available:

Method	Description
`select`	A standard method. The supporting module is built automatically on platforms that do not have more efficient methods. The `--with-select_module` and `--without-select_module` build options can be used to forcibly enable or disable the building of this module.
`poll`	A standard method. The supporting module is built automatically on platforms that do not have more efficient methods. The `--with-poll_module` and `--without-poll_module` build options can be used to forcibly enable or disable the building of this module.
`kqueue`	An efficient method available on FreeBSD 4.1+, OpenBSD 2.9+, NetBSD 2.0, and macOS.
`epoll`	An efficient method available on Linux 2.6+.
`/dev/poll`	An efficient method available on Solaris 7 11/99+, HP/UX 11.22+ (eventport), IRIX 6.5.15+, and Tru64 UNIX 5.1A+.
`eventport`	The `event ports` method is available on Solaris 10+. (Due to known issues, using the `/dev/poll` method is recommended instead.)

HTTP request processing

An HTTP request goes through a series of phases, where a specific type of processing is performed at each phase.

`Post-read`	The initial phase. The *RealIP* module is invoked during this phase.
`Server-rewrite`	The phase where directives from the *Rewrite* module, defined in a `server` block (but outside a `location` block), are processed.
`Find-config`	A special phase where a *location* is selected based on the request URI.
`Rewrite`	Similar to the `Server-rewrite` phase, but it applies to *rewrite* rules defined within the location block selected in the previous phase.
`Post-rewrite`	A special phase where the request is redirected to a new location, as in the `Find-config` phase, if its URI was modified during the `Rewrite` phase.
`Preaccess`	During this phase, standard Angie modules like *Limit Req* register their handlers.
`Access`	The phase where the client's authorization to make the request is verified, typically by invoking standard Angie modules such as *Auth Basic*.
`Post-access`	A special phase where the *satisfy any* directive is processed.
`Precontent`	Standard module directives, such as *try_files* and *mirror*, register their handlers during this phase.
`Content`	The phase where the response is usually generated. Multiple standard Angie modules register their handlers at this stage, including *Index*. The *proxy_pass*, *fastcgi_pass*, *uwsgi_pass*, *scgi_pass* and *grpc_pass* directives are also handled here. Handlers are called sequentially until one of them produces the output.
`Log`	The final phase, where request logging is performed. Currently, only the *Log* module registers its handler at this stage for access logging.

TCP/UDP session processing

A TCP/UDP session from a client goes through a series of phases, where a specific type of processing is performed at each phase:

`Post-accept`	The initial phase after accepting a client connection. The *RealIP* module is invoked at this phase.
`Pre-access`	A preliminary phase for checking access. The *Set* modules are invoked during this phase.
`Access`	The phase for limiting client access before actual data processing. The *Access* module is invoked at this stage.
`SSL`	The phase where TLS/SSL termination occurs. The *SSL* module is invoked during this phase.
`Preread`	The phase for reading initial bytes of data into the *preread buffer* to allow modules such as *SSL Preread* to analyze the data before processing.
`Content`	A mandatory phase where the data is actually processed, typically involving the *Return* module to send a response to the client. The *proxy_pass* directive is also handled here.
`Log`	The final phase where the outcome of client session processing is recorded. The *Log* module is invoked at this phase.

Processing requests

Virtual server selection

Initially, a connection is created within the context of a default server. The server name can then be determined in the following stages of request processing, each of which is involved in the selection of server configuration:

- During the SSL handshake, in advance, according to the SNI.
- After processing the request line.
- After processing the `Host` header field.

If the server name is not determined after processing the request line or the `Host` header field, Angie will use an empty name as the server name.

At each of these stages, different server configurations may be applied. Therefore, certain directives should be specified with caution:

- In the case of the *ssl_protocols* directive, the protocol list is set by the OpenSSL library before the server configuration is applied according to the name requested through SNI. As a result, protocols should only be specified for the default server.
- The *client_header_buffer_size* and *merge_slashes* directives are applied before reading the request line. Therefore, these directives use either the default server configuration or the server configuration chosen by SNI.
- In the case of the *ignore_invalid_headers*, *large_client_header_buffers*, and *underscores_in_headers* directives, which are involved in processing request header fields, the server configuration additionally depends on whether it was updated according to the request line or the `Host` header field.
- An error response is handled using the *error_page* directive in the server that is currently processing the request.

Name-based virtual servers

Angie first determines which server should handle the request. Consider a simple configuration where all three virtual servers listen on port 80:

```
server {

    listen 80;
    server_name example.org www.example.org;
    # ...
}

server {

    listen 80;
    server_name example.net www.example.net;
    #  ...
}

server {

    listen 80;
    server_name example.com www.example.com;
    #  ...
}
```

In this configuration, Angie determines which server should handle the request based solely on the `Host` header field. If the value of this header does not match any server name or if the request does not contain this header field, Angie will route the request to the default server for this port. In the configuration above, the default server is the first one — which is Angie's standard default behavior. It can also be explicitly specified which server should be the default using the `default_server` parameter in the *listen* directive:

```
server {

    listen 80 default_server;
    server_name example.net www.example.net;
    #  ...
}
```

> **Note**
>
> Note that the default server is a property of the listen socket, not of the server name.

Internationalized names

Internationalized domain names (IDNs) should be specified using an ASCII (Punycode) representation in the *server_name* directive:

```
server {

    listen 80;
    server_name xn--e1afmkfd.xn--80akhbyknj4f; # пример.испытание
    #    ...
}
```

Preventing requests with undefined server names

If requests without the `Host` header field should not be allowed, a server that simply drops such requests can be defined:

```
server {

    listen 80;
    server_name "";
    return 444;
}
```

In this configuration, the server name is set to an empty string, which matches requests without the `Host` header field. A special non-standard code 444 is then returned, which closes the connection.

Combining name-based and IP-based virtual servers

Let's examine a more complex configuration where some virtual servers listen on different addresses:

```
server {

    listen 192.168.1.1:80;
    server_name example.org www.example.org;
    #  ...
}

server {

    listen 192.168.1.1:80;
    server_name example.net www.example.net;
    #  ...
}

server {

    listen 192.168.1.2:80;
    server_name example.com www.example.com;
    #  ...
}
```

In this configuration, Angie first tests the IP address and port of the request against the *listen* directives

of the *server* blocks. It then tests the `Host` header field of the request against the *server_name* entries of the *server* blocks that matched the IP address and port. If the server name is not found, the request will be processed by the default server. For example, a request for `www.example.com` received on port 192.168.1.1:80 will be handled by the default server for that port — i.e., by the first server — since `www.example.com` is not defined for this port.

As previously mentioned, a default server is a property of the listen port, and different default servers may be defined for different ports:

```
server {

    listen 192.168.1.1:80;
    server_name example.org www.example.org;
    #  ...
}

server {

    listen 192.168.1.1:80 default_server;
    server_name example.net www.example.net;
    #  ...
}

server {

    listen 192.168.1.2:80 default_server;
    server_name example.com www.example.com;
    #  ...
}
```

Choosing locations

Consider a simple PHP website configuration:

```
server {

    listen 80;
    server_name example.org www.example.org;
    root /data/www;

    location / {

        index index.html index.php;
    }

    location ~* \.(gif|jpg|png)$ {

        expires 30d;
    }

    location ~ \.php$ {

        fastcgi_pass localhost:9000;
        fastcgi_param SCRIPT_FILENAME
        $document_root$fastcgi_script_name;
        include fastcgi_params;
    }
}
```

Angie first searches for the most specific prefix `location` given by literal strings, regardless of their listed order. In the configuration above, the only prefix location is `location /`, which matches any request and will be used as a last resort. Angie then checks locations defined by regular expressions in the order they appear in the configuration file. The first matching expression stops the search, and Angie will use that `location`. If no regular expression matches a request, Angie will use the most specific prefix `location` found earlier.

> **Note**
>
> Locations of all types test only the URI part of the request line, excluding arguments. This is because arguments in the query string can be specified in various ways, for example:
>
> - `/index.php?user=john&page=1`
> - `/index.php?page=1&user=john`
>
> Additionally, query strings may contain any number of parameters:
>
> - `/index.php?page=1&something+else&user=john`

Now let's look at how requests would be processed in the configuration above:

- The request `/logo.gif` is first matched by the prefix `location /` and then by the regular expression `.(gif|jpg|png)$`. Therefore, it is handled by the latter location. Using the directive `root /data/www`, the request is mapped to the file `/data/www/logo.gif`, and the file is sent to the client.
- The request `/index.php` is also initially matched by the prefix `location /` and then by the regular expression `.(php)$`. Consequently, it is handled by the latter location, and the request is passed to a FastCGI server listening on `localhost:9000`. The *fastcgi_param* directive sets the FastCGI parameter `SCRIPT_FILENAME` to `/data/www/index.php`, and the FastCGI server executes the file. The variable *$document_root* is set to the value of the `root` directive, and the variable *$fastcgi_script_name* is set to the request URI, i.e., `/index.php`.
- The request `/about.html` is matched only by the prefix `location /`, so it is handled in this location. Using the directive `root /data/www`, the request is mapped to the file `/data/www/about.html`, and the file is sent to the client.

Handling the request `/` is more complex. It is matched only by the prefix *location /*, so it is handled by this location. The *index* directive then tests for the existence of index files according to its parameters and the `root /data/www` directive. If the file `/data/www/index.html` does not exist but the file `/data/www/index.php` does, the directive performs an internal redirect to `/index.php`, and Angie searches the locations again as if the request had been sent by a client. As previously mentioned, the redirected request will eventually be handled by the FastCGI server.

Proxying and Load Balancing

One common use of Angie is to set it up as a proxy server. In this role, Angie receives requests, forwards them to the proxied servers, retrieves responses from those servers, and sends the responses back to the clients.

A simple proxy server:

```
server {

    location / {

        proxy_pass http://backend:8080;
    }
```

The *proxy_pass* directive instructs Angie to pass client requests to the backend `backend:8080` (the proxied server). There are many additional *directives* available for further configuring a proxy connection.

FastCGI Proxying

Angie can be used to route requests to FastCGI servers that run applications built with various frameworks and programming languages, such as PHP.

The most basic Angie configuration for working with a FastCGI server involves using the *fastcgi_pass* directive instead of the *proxy_pass* directive, along with *fastcgi_param* directives to set parameters passed to the FastCGI server. Suppose the FastCGI server is accessible on `localhost:9000`. In PHP, the `SCRIPT_FILENAME` parameter is used to determine the script name, and the `QUERY_STRING` parameter is used to pass request parameters. The resulting configuration would be:

```
server {

    location / {

        fastcgi_pass localhost:9000;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_param QUERY_STRING $query_string;
    }

    location ~ \.(gif|jpg|png)$ {

        root /data/images;
    }
}
```

This configuration sets up a server that routes all requests, except those for static images, to the proxied server operating on `localhost:9000` via the FastCGI protocol.

WebSocket Proxying

To upgrade a connection from HTTP/1.1 to WebSocket, the protocol switch mechanism available in HTTP/1.1 is used.

However, there is a subtlety: since the `Upgrade` header is a hop-by-hop header, it is not passed from the client to the proxied server. With forward proxying, clients may use the CONNECT method to circumvent this issue. This approach does not work with reverse proxying, as clients are unaware of any proxy servers, and special processing on the proxy server is required.

Angie implements a special mode of operation that allows setting up a tunnel between a client and a proxied server if the proxied server returns a response with code 101 (Switching Protocols), and the client requests a protocol switch via the `Upgrade` header in the request.

As mentioned, hop-by-hop headers, including `Upgrade` and `Connection`, are not passed from the client to the proxied server. Therefore, for the proxied server to be aware of the client's intention to switch to the WebSocket protocol, these headers must be explicitly passed:

```
location /chat/ {

    proxy_pass http://backend;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
}
```

A more sophisticated example demonstrates how the value of the `Connection` header field in a request to the proxied server depends on the presence of the `Upgrade` field in the client request header:

```
http {

    map $http_upgrade $connection_upgrade {
```

```
        default upgrade;
        '' close;
    }

    server {

        ...

        location /chat/ {

            proxy_pass http://backend;
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection $connection_upgrade;
        }
    }
}
```

By default, the connection will be closed if the proxied server does not transmit any data within 60 seconds. This timeout can be increased using the *proxy_read_timeout* directive. Alternatively, the proxied server can be configured to periodically send WebSocket ping frames to reset the timeout and check if the connection is still active.

Load Balancing

Load balancing across multiple application instances is a widely used technique to optimize resource utilization, maximize throughput, reduce latency, and ensure fault-tolerant configurations.

Angie can be used as a highly efficient HTTP load balancer to distribute traffic to multiple application servers, thereby enhancing the performance, scalability, and reliability of web applications.

The simplest configuration for load balancing with Angie might look like this:

```
http {

    upstream myapp1 {

        server srv1.example.com;
        server srv2.example.com;
        server srv3.example.com;
    }

    server {

        listen 80;

        location / {

            proxy_pass http://myapp1;
        }
    }
}
```

In the example above, three instances of the same application are running on `srv1` through `srv3`. When a load balancing method is not explicitly configured, it defaults to round-robin. Other supported load balancing mechanisms include: *weight*, *least_conn*, and *ip_hash*. The reverse proxy implementation in Angie also supports in-band (or passive) server health checks. These are configured using the *max_fails* and *fail_timeout* directives within the *server* block in the *upstream* context.

Logging

> **Note**
>
> In addition to the options listed here, you can also enable the *debugging log*.

Syslog

The *error_log* and *access_log* directives support logging to `syslog`. The following parameters are used to configure logging to `syslog`:

`server=`*address*	Specifies the address of a `syslog` server. The address can be a domain name or an IP address, with an optional port, or a UNIX domain socket path specified after the `"unix:"` prefix. If the port is not specified, UDP port 514 is used. If a domain name resolves to multiple IP addresses, the first resolved address is used.
`facility=`*string*	Sets the facility for `syslog` messages, as defined in RFC 3164. Possible facilities include: `"kern"`, `"user"`, `"mail"`, `"daemon"`, `"auth"`, `"intern"`, `"lpr"`, `"news"`, `"uucp"`, `"clock"`, `"authpriv"`, `"ftp"`, `"ntp"`, `"audit"`, `"alert"`, `"cron"`, `"local0"..."local7"`. The default is `"local7"`.
`severity=`*string*	Defines the severity level of `syslog` messages for *access_log*, as specified in RFC 3164. Possible values are the same as those for the second parameter (level) of the *error_log* directive. The default is `"info"`. The severity of error messages is determined by Angie, so this parameter is ignored in the *error_log* directive.
`tag=`*string*	Sets the tag for `syslog` messages. The default tag is `"angie"`.
`nohostname`	Disables the addition of the :sa,p:*hostname* field in the `syslog` message header.

Example syslog configuration:

```
error_log syslog:server=192.168.1.1 debug;

access_log syslog:server=unix:/var/log/angie.sock,nohostname;
access_log syslog:server=[2001:db8::1]:12345,facility=local7,tag=angie,severity=info␣
↪combined;
```

> **Note**
>
> Syslog entries are reported no more than once per second to prevent flooding.

3.2 Indexes, References

These summaries provide reference information on built-in modules, examples of their configuration, as well as supported directives and variables.

3.2.1 Built-in Modules

This guide describes the built-in modules of Angie, provides configuration examples, lists their directives and parameters, as well as built-in variables.

Core Module

The module provides essential functionality and configuration directives necessary for the basic operation of the server, and handles critical tasks such as managing worker processes, configuring event-driven models, and processing incoming connections and requests. It includes key directives for setting up the main process, error logging, and controlling the behavior of the server at a low level.

Configuration Example

```
user www www;
worker_processes 2;

error_log /var/log/error.log info;

events {

    use kqueue; worker_connections 2048;
}
```

Directives

accept_mutex

Syntax	`accept_mutex on	off;`
Default	`accept_mutex off;`
Context	events

When `accept_mutex` is enabled, worker processes will accept new connections in turn. Without this setting, all worker processes are notified of new connections, which can lead to inefficient use of system resources if the volume of new connections is low.

> **Note**
>
> There is no need to enable `accept_mutex` on systems that support the `EPOLLEXCLUSIVE` flag or when using the *reuseport* directive.

accept_mutex_delay

Syntax	`accept_mutex_delay` *time*;
Default	`accept_mutex_delay 500ms;`
Context	events

If *accept_mutex* is enabled, this directive specifies the maximum time a worker process will wait to continue accepting new connections while another worker process is already handling new connections.

daemon

Syntax	`daemon on	off;`
Default	`daemon on;`
Context	main

Determines whether Angie should run as a daemon. This is primarily used during development.

debug_connection

Syntax	`debug_connection` *address*	*CIDR*	`unix:;`
Default	—
Context	events

Enables debugging logs for specific client connections. Other connections will use the logging level set by the *error_log* directive. You can specify connections by IPv4 or IPv6 address, network, or hostname. For connections using UNIX domain sockets, use the `unix:` parameter to enable debugging logs.

```
events {

    debug_connection 127.0.0.1;
    debug_connection localhost;
    debug_connection 192.0.2.0/24;
    debug_connection ::1;
    debug_connection 2001:0db8::/32;
    debug_connection unix:;
    #  ...
}
```

Important

For this directive to work, Angie must be built with *debugging log* enabled.

debug_points

Syntax	`debug_points abort	stop;`
Default	—
Context	main

This directive is used for debugging.

When an internal error occurs, such as a socket leak during worker process restarts, enabling `debug_points` will either create a core file (`abort`) or stop the process (`stop`) for further analysis with a system debugger.

env

Syntax	`env` *variable*[=*value*];
Default	`env TZ;`
Context	main

By default, Angie removes all environment variables inherited from its parent process except for the `TZ` variable. This directive allows you to preserve some inherited variables, modify their values, or create new environment variables.

These variables are then:

- Inherited during a *live upgrade of an executable file*
- Used by the *Perl* module
- Available to worker processes

Note that controlling system libraries in this way may not always be effective, as libraries often check variables only during initialization, which occurs before this directive takes effect. The `TZ` variable is always inherited and accessible to the *Perl* module unless explicitly configured otherwise.

Example:

```
env MALLOC_OPTIONS;
env PERL5LIB=/data/site/modules;
env OPENSSL_ALLOW_PROXY_CERTS=1;
```

Note

The `ANGIE` environment variable is used internally by Angie and should not be set directly by the user.

error_log

Syntax	`error_log` *file* [*level*];
Default	`error_log logs/error.log error;` (the path depends on the `--error-log-path` build option)
Context	main, http, mail, stream, server, location

Configures logging, allowing multiple logs to be specified at the same configuration level. If a log file is not explicitly defined at the `main` configuration level, the default file will be used.

The first parameter specifies the file to store the log. The special value `stderr` selects the standard error stream. To configure logging to *syslog*, use the `"syslog:"` prefix. To log to a *cyclic memory buffer*, use the `"memory:"` prefix followed by the buffer size; this is typically used for debugging.

The second parameter sets the logging level, which can be one of the following: `debug`, `info`, `notice`, `warn`, `error`, `crit`, `alert`, or `emerg`. These levels are listed in order of increasing severity. Setting a log level will capture messages of equal and higher severity:

Setting	Levels Captured
`debug`	`debug`, `info`, `notice`, `warn`, `error`, `crit`, `alert`, `emerg`
`info`	`info`, `notice`, `warn`, `error`, `crit`, `alert`, `emerg`
`notice`	`notice`, `warn`, `error`, `crit`, `alert`, `emerg`
`warn`	`warn`, `error`, `crit`, `alert`, `emerg`
`error`	`error`, `crit`, `alert`, `emerg`
`crit`	`crit`, `alert`, `emerg`
`alert`	`alert`, `emerg`
`emerg`	`emerg`

If this parameter is omitted, `error` is used as the default logging level.

Important

For the `debug` logging level to work, Angie must be built with *debugging log* enabled.

events

Syntax	`events` { ... };
Default	—
Context	main

Provides the configuration file context for directives that affect connection processing.

include

Syntax	`include` *file* | *mask*;
Default	—
Context	any

Includes another file, or files that match the specified *mask*, into the configuration. The included files must contain syntactically correct directives and blocks.

Example:

```
include mime.types;
include vhosts/*.conf;
```

load_module

Syntax	`load_module` *file*;
Default	—
Context	main

Loads a dynamic module from the specified file. If a relative path is provided, it is interpreted based on the `--prefix` build option. To verify the path:

```
$ sudo angie -V
```

Example:

```
load_module modules/ngx_mail_module.so;
```

lock_file

Syntax	`lock_file` *file*;
Default	`lock_file logs/angie.lock;` (the path depends on the `--lock-path` build option)
Context	main

Angie uses a locking mechanism to implement *accept_mutex* and serialize access to shared memory. On most systems, locks are managed using atomic operations, making this directive unnecessary. On certain systems, however, an alternative *lock file* mechanism is used. This directive sets a prefix for lock file names.

master_process

Syntax	`master_process on` | `off`;
Default	`master_process on;`
Context	main

Determines whether worker processes are started. This directive is intended for Angie developers.

multi_accept

Syntax	`multi_accept on	off;`
Default	`multi_accept off;`
Context	events

`on`	A worker process will accept all new connections simultaneously.
`off`	A worker process will accept one new connection at a time.

Note

This directive is ignored if the *kqueue* connection processing method is used, as it provides the number of new connections ready to be accepted.

pcre_jit

Syntax	`pcre_jit on	off;`
Default	`pcre_jit off;`
Context	main

Enables or disables "just-in-time compilation" (PCRE JIT) for regular expressions known at the time of configuration parsing.

PCRE JIT can significantly accelerate regular expression processing.

Important

JIT is available in PCRE libraries from version 8.20, provided they are built with the `--enable-jit` configuration option. When Angie is built with the PCRE library (`--with-pcre=`), JIT support is enabled using the `--with-pcre-jit` option.

pid

Syntax	`pid` *file* `| off;`
Default	`pid logs/angie.pid;` (the path depends on the `--pid-path` build option)
Context	main

Specifies the *file* that will store the ID of the Angie main process. The file is created atomically, which ensures its contents are always correct. The `off` setting disables the creation of this file.

Note

If the *file* setting is modified during reconfiguration but points to a symlink of the previous PID file, the file will not be recreated.

ssl_engine

Syntax	`ssl_engine` *device*;
Default	—
Context	main

Specifies the name of the hardware SSL accelerator.

ssl_object_cache_inheritable

Syntax	`ssl_object_cache_inheritable on | off;`
Default	`ssl_object_cache_inheritable on;`
Context	main

If enabled, SSL objects (SSL certificates, secret keys, trusted CA certificates, CRL lists) are inherited across configuration reloads.

SSL objects loaded from files are inherited if their modification time and file index have not changed since the previous configuration load. Secret keys specified as `engine:name:id` are never inherited, while secret keys specified as `data:value` are always inherited.

SSL objects loaded from variables cannot be inherited.

Example:

```
ssl_object_cache_inheritable on;

http {
    server {
        ssl_certificate     example.com.crt;
        ssl_certificate_key example.com.key;
    }
}
```

thread_pool

Syntax	`thread_pool` *name* `threads=`*number* [`max_queue=`*number*];
Default	`thread_pool default threads=32 max_queue=65536;`
Context	main

Defines the *name* and parameters of a thread pool used for multi-threaded file reading and sending, *without blocking* worker processes.

The `threads` parameter specifies the number of threads in the pool.

If all threads are busy, new tasks will wait in the queue. The `max_queue` parameter limits the number of tasks that can wait in the queue. By default, the queue can hold up to 65536 tasks. When the queue overflows, new tasks are completed with an error.

timer_resolution

Syntax	`timer_resolution` *interval*;
Default	—
Context	main

Reduces timer resolution in worker processes, thereby decreasing the frequency of `gettimeofday()` system calls. By default, `gettimeofday()` is called each time kernel events are queried. With reduced resolution, it is called only once per the specified interval.

Example:

```
timer_resolution 100ms;
```

The internal implementation of the interval depends on the method used:

- The `EVFILT_TIMER` filter if *kqueue* is used.
- The `timer_create()` function if *eventport* is used.
- The `setitimer()` function otherwise.

use

Syntax	`use` *method*;
Default	—
Context	events

Specifies the *connection processing method* to use. Typically, there is no need to specify it explicitly, as Angie defaults to the most efficient method.

user

Syntax	`user` *user* [*group*];
Default	`user <--user build option> <--group build option>;`
Context	main

Defines the user and group credentials for worker processes (see also the build options). If only the user is set, the specified user name is also used for the group.

worker_aio_requests

Syntax	`worker_aio_requests` *number*;
Default	`worker_aio_requests 32;`
Context	events

When using *aio* with the *epoll* connection processing method, sets the maximum *number* of outstanding asynchronous I/O operations for a single worker process.

worker_connections

Syntax	`worker_connections` *number*;
Default	`worker_connections 512;`
Context	events

Sets the maximum number of simultaneous connections a worker process can open.

Note that this number includes all connections, such as those with proxied servers, not just client connections. Additionally, the actual number of simultaneous connections cannot exceed the system's limit on open files, which can be adjusted using *worker_rlimit_nofile*.

worker_cpu_affinity

Syntax	`worker_cpu_affinity` *cpumask* ...; `worker_cpu_affinity` auto [*cpumask*];
Default	—
Context	main

Binds worker processes to specific sets of CPUs. Each CPU set is represented by a bitmask indicating allowed CPUs. A separate set should be defined for each worker process. By default, worker processes are not bound to any specific CPUs.

For example:

```
worker_processes    4;
worker_cpu_affinity 0001 0010 0100 1000;
```

This configuration binds each worker process to a separate CPU.

Alternatively:

```
worker_processes    2;
worker_cpu_affinity 0101 1010;
```

This binds the first worker process to CPU0 and CPU2, and the second worker process to CPU1 and CPU3. This setup is suitable for hyper-threading.

The special value `auto` allows automatic binding of worker processes to available CPUs:

```
worker_processes auto;
worker_cpu_affinity auto;
```

The optional mask parameter can be used to limit the CPUs available for automatic binding:

```
worker_cpu_affinity auto 01010101;
```

Important

This directive is only available on FreeBSD and Linux.

worker_priority

Syntax	`worker_priority` *number*;
Default	`worker_priority 0;`
Context	main

Defines the scheduling priority for worker processes, similar to the `nice` command: a negative *number* indicates higher priority. The allowed range typically varies from -20 to 20.

Example:

```
worker_priority -10;
```

worker_processes

Syntax	`worker_processes` *number* | `auto`;
Default	`worker_processes 1;`
Context	main

Defines the number of worker processes.

The optimal value depends on various factors, including the number of CPU cores, the number of hard disk drives, and the load pattern. If unsure, starting with the number of available CPU cores is recommended. The value `auto` attempts to autodetect the optimal worker process number.

worker_rlimit_core

Syntax	`worker_rlimit_core` *size*;
Default	—
Context	main

Changes the limit on the maximum size of a core file (`RLIMIT_CORE`) for worker processes. This allows increasing the limit without restarting the main process.

worker_rlimit_nofile

Syntax	`worker_rlimit_nofile` *number*;
Default	—
Context	main

Changes the limit on the maximum number of open files (`RLIMIT_NOFILE`) for worker processes. This allows increasing the limit without restarting the main process.

worker_shutdown_timeout

Syntax	`worker_shutdown_timeout` *time*;
Default	—
Context	main

Configures a timeout for the graceful shutdown of worker processes. Once the *time* expires, Angie will attempt to close all active connections to complete the shutdown process.

A graceful shutdown is initiated by sending a *QUIT signal* to the master process, which tells the worker processes to stop accepting new connections while allowing existing connections to complete. Workers continue to process active requests until they finish, after which they exit cleanly. If the connections remain open beyond `worker_shutdown_timeout`, Angie will forcefully close those connections to finalize the shutdown. Also, keepalive connections from clients are closed only if they remain inactive for at least the duration of *lingering_timeout*.

working_directory

Syntax	`working_directory` *directory*;
Default	—
Context	main

Defines the current working directory for a worker process. This is primarily used for writing core files, so the worker process must have write permissions for the specified directory.

HTTP Module

Access

The module controls access to server resources based on client IP addresses or networks. It allows to permit or block specific IPs, IP ranges, or UNIX domain sockets to enhance security by restricting access to sensitive areas of a website or application.

Access can also be restricted by using a password with the *Auth Basic* module or based on the result of a subrequest with the *Auth Request* module. To apply both address and password restrictions at the same time, use the *satisfy* directive.

Configuration Example

```
location / {

    deny 192.168.1.1;
    allow 192.168.1.0/24;
    allow 10.1.1.0/16;
    allow 2001:0db8::/32;
    deny all;
}
```

Rules are evaluated sequentially until a match is found. In this example, access is allowed only for the IPv4 networks `10.1.1.0/16` and `192.168.1.0/24`, excluding the specific address `192.168.1.1`, and for the IPv6 network `2001:0db8::/32`. When there are many rules, it is preferable to use variables from the *Geo* module.

Directives

allow

Syntax	`allow` *address* | *CIDR* | `unix:` | `all`;
Default	—
Context	http, server, location, limit_except

Allows access for a specified network or address. The special value `all` means all client IPs.

Added in version 1.5.1: The special value `unix:` allows access for any UNIX domain sockets.

deny

Syntax	`deny` *address* | *CIDR* | `unix:` | `all`;
Default	—
Context	http, server, location, limit_except

Denies access for a specified network or address. The special value `all` means all client IPs.

Added in version 1.5.1: The special value `unix:` denies access for any UNIX domain sockets.

ACME

Provides automatic certificate retrieval using the ACME protocol.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-http_acme_module` build option.

In packages and images from our repos, the module is included in the build.

Configuration Example

Examples of configuration and instructions for setup can be found in the *ACME Configuration* section.

Directives

acme

Syntax	`acme` *name*;
Default	—
Context	server

For all domains specified in the *server_name* directives of all *server* blocks that refer to the *acme_client* called *name*, a single certificate will be obtained; if the `server_name` configuration changes, the certificate is renewed to reflect the updates.

Each time Angie starts, certificates are requested for all domains that are missing a valid certificate. Possible reasons include expired certificates; missing or unreadable files; configuration changes for a certificate.

Note

Currently, domains specified with regular expressions are not supported and will be ignored.

Wildcard domains are supported only with `challenge=dns` setting in `acme_client`.

This directive can be specified multiple times to load certificates of different types, for example, RSA and ECDSA:

```
server {

    listen 443 ssl;
    server_name example.com www.example.com;

    ssl_certificate $acme_cert_rsa;
    ssl_certificate_key $acme_cert_key_rsa;

    ssl_certificate $acme_cert_ecdsa;
    ssl_certificate_key $acme_cert_key_ecdsa;

    acme rsa;
    acme ecdsa;
}
```

acme_client

Syntax	`acme_client` *name uri* [`enabled=on` | `off`] [`key_type=`*type*] [`key_bits=`*number*] [`email=`*email*] [`max_cert_size=`*number*] [`renew_before_expiry=`*time*] [`renew_on_load`] [`retry_after_error=`off|*time*] [`challenge=dns` | `http`] [`account_key=`*file*];
Default	—
Context	http

Defines an ACME client under a globally unique *name*. It must be valid for a file directory, is a string with variables and will be treated as case insensitive.

Tip

The client name defined here identifies it in the Angie configuration, allowing you to match `acme_client`, *acme* directives, and module *variables* that use this name; don't confuse it with your domain or server name.

The second mandatory parameter is the *uri* of the ACME directory. For example, the URI of Let's Encrypt ACME directory is listed as https://acme-v02.api.letsencrypt.org/directory.

For this directive to work, a *resolver* must be configured in the same context.

Note

For testing purposes, certificate authorities usually provide separate staging environments. For example, Let's Encrypt staging environment is currently https://acme-staging-v02.api.letsencrypt.org/directory.

`enabled`	Enables or disables certificate renewal for the client; this is useful, for example, for temporarily suspending without removing the client from the configuration. Default: `on`.
`key_type`	The type of private key algorithm for the certificate. Valid values: `rsa`, `ecdsa`. Default: `ecdsa`.
`key_bits`	Number of bits in the certificate key. Default: 256 for `ecdsa`, 2048 for `rsa`.
`email`	Optional email address for feedback; used when creating an account on the CA server.
`max_cert_size`	Specifies the maximum allowed size of a new certificate file in bytes to reserve space for a new certificate in shared memory; the more domains the certificate is requested for, the more space is required. If a certificate already exists at startup but its size exceeds the value of `max_cert_size`, the `max_cert_size` value is dynamically increased to match the size of the existing certificate file. If the size of a renewed certificate exceeds `max_cert_size`, the renewal will fail with an error. Default: `8192`.
`renew_before_exp`	*Time* before the certificate expires when certificate renewal should start. Default: `30d`.
`renew_on_load`	Specifies that the certificate should be forcibly renewed each time the configuration is loaded.
`retry_after_erro`	*Time* to retry when a certificate couldn't be obtained. If set to `off`, the client won't retry to obtain the certificate after a failure. Default: `2h`.
`challenge`	Specifies the verification type for the ACME client. Allowed values: `dns`, `http`. Default: `http`.
`account_key`	Specifies the full path to a file containing a key in PEM format. This is useful if you want to use an existing account key instead of generating one automatically, or if you need to use one key for multiple ACME clients. Supported key types: • RSA keys with lengths that are multiples of 8, ranging from 2048 to 8192 bits. • ECDSA keys with lengths of 256, 384, or 521 bits. When specifying the `account_key` parameter, make sure the key file actually exists. If the file is missing, Angie will attempt to create it at the specified path. Note that keys for ACME clients are created in the order the corresponding clients are mentioned in the configuration in *acme_client*, *acme*, or *acme_hook* directives. Therefore, if one client should use a key created for another, that other client must appear earlier in the configuration. Additionally, keys are only created for clients that have the `enabled=on` parameter set.

acme_client_path

Syntax	`acme_client_path` *path*;
Default	—
Context	http

Overrides the *path* to the directory for storing certificates and keys, specified at build time with the `--http-acme-client-path` build option.

acme_dns_port

Syntax	`acme_dns_port` *port* | *ip:port* | *[ip6]:port*;
Default	`acme_dns_port 53;`
Context	http

Specifies the port that the module uses to handle DNS queries from the ACME server over UDP. The port number must be between 1 and 65535.

Specifying an IP address along with the optional port is also supported. Both IPv4 addresses in the form `ip:port` and IPv6 addresses in the form `[ip6]:port` can be used:

```
acme_dns_port 8053;
acme_dns_port 127.0.0.1;
acme_dns_port [::1];
```

To use a port number of 1024 or lower, Angie must run with *superuser* privileges.

acme_hook

Syntax	`acme_hook` *name* [*uri*];
Default	—
Context	location

The directive links the server to the specified ACME client. Handler (hook) calls implemented by an external service are made through the `location` context where it is defined.

name	Specifies the associated ACME client.
uri	A string with variables; defines the request string for handler calls. Default is `/`.

For example, the following configuration passes the values of *hook variables* to a FastCGI application through the request string:

```
acme_hook example uri=/acme_hook/$acme_hook_name?domain=$acme_hook_domain&key=$acme_
↪hook_keyauth;
fastcgi_param REQUEST_URI $request_uri;
fastcgi_pass ...;
```

Built-in Variables

`$acme_cert_<name>`

Contents of the last certificate file (if any) obtained by the client with this *name*.

`$acme_cert_key_<name>`

Contents of the certificate key file used by the client with this *name*.

Important

The certificate file is available only if the ACME client has received at least one certificate, whereas the key file is available immediately after startup.

`$acme_hook_challenge`

The verification type. Possible values: `dns`, `http`.

`$acme_hook_client`

The name of the ACME client initiating the request.

`$acme_hook_domain`

The domain being verified. If it is a wildcard domain, it will be passed without the `*.` prefix.

`$acme_hook_keyauth`

The authorization string:

- For DNS verification, it is used as the value of the TXT record, whose name is formed as `_acme-challenge.` + $acme_hook_domain + ..
- For HTTP verification, this string must be used as the content of the response requested by the ACME server.

`$acme_hook_name`

The name of the hook. For different verification types, it may have different values and meanings:

Value	Meaning for DNS verification	Meaning for HTTP verification
`add` (adding the hook)	The corresponding TXT record must be added to the DNS configuration.	The appropriate response to the HTTP request must be prepared.
`remove` (removing the hook)	The TXT record can be removed from the DNS configuration.	This HTTP request is no longer relevant; a previously created file with the authorization string can be removed.

`$acme_hook_token`

The verification token. For HTTP verification, it is used as the name of the requested file: `/.well-known/acme-challenge/` + `$acme_hook_token`.

Addition

The module is a filter that adds text before and after a response.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-http_addition_module` build option.

In packages and images from our repos, the module is included in the build.

Configuration Example

```
location / {
    add_before_body /before_action;
    add_after_body  /after_action;
}
```

Directives

add_before_body

Syntax	`add_before_body` *uri*;
Default	—
Context	http, server, location

Adds the text returned as a result of processing a given subrequest before the response body. An empty string ("") as a parameter cancels addition inherited from the previous configuration level.

add_after_body

Syntax	`add_after_body` *uri*;
Default	—
Context	http, server, location

Adds the text returned as a result of processing a given subrequest after the response body. An empty string ("") as a parameter cancels addition inherited from the previous configuration level.

addition_types

Syntax	`addition_types` mime-*type* ...;
Default	`addition_types text/html;`
Context	http, server, location

Allows adding text in responses with the specified MIME types, in addition to "text/html". The special value "*" matches any MIME type.

API

The module provides HTTP RESTful interface for accessing in JSON format basic information about a web server instance, as well as metrics of client connections, shared memory zones, DNS queries, HTTP requests, HTTP responses cache, TCP/UDP sessions of *Stream Module*, and zones of *Limit Conn*, *Limit Conn*, *Limit Req* and *Upstream* modules.

The API accepts `GET` and `HEAD` HTTP requests; other request methods cause an error:

```
{
    "error": "MethodNotAllowed",
    "description": "The POST method is not allowed for the requested API element \"/\
↪"."
}
```

The Angie PRO API has a *dynamic configuration* that allows updating the settings without reloading the configuration or restarting Angie itself; currently, it enables configuring individual peers in an upstream.

Directives

api

Syntax	`api` *path*;
Default	—
Context	location

Enables HTTP RESTful interface in `location`.

path parameter is mandatory and works similar to the *alias* directive, but operates over API tree, rather than filesystem hierarchy.

When specified in a prefix location:

```
location /stats/ {
    api /status/http/server_zones/;
}
```

the part of request URI matching particular prefix location `/stats/` will be replaced by the path `/status/http/server_zones/` in the directive parameter. For example, on a request of `/stats/foo/` the `/status/http/server_zones/foo/` API element will be accessed.

Also it's possible to use variables: *api /status/$module/server_zones/$name/* and the directive can also be specified inside a regexp location:

```
location ~^/api/([^/]+)/(.*)$ {
    api /status/http/$1_zones/$2;
}
```

here, similar to *alias*, parameter defines the whole path to the API element. E.g., from `/api/location/data/` the following positional variables will be populated:

```
$1 = "location"
$2 = "data/"
```

which results after interpolation in `/status/http/location_zones/data/`.

Note

In Angie PRO, you can decouple the *dynamic configuration API* from the immutable *metrics API* that reflects current state:

```
location /config/ {
    api /config/;
}

location /status/ {
    api /status/;
}
```

Overall, it serves for precise configuration of API access rights, e.g.:

```
location /status/ {
    api /status/;

    allow 127.0.0.1;
    deny  all;
}
```

Or:

```
location /blog/requests/ {
    api /status/http/server_zones/blog/requests/;

    auth_basic           "blog";
    auth_basic_user_file conf/htpasswd;
}
```

api_config_files

Syntax	`api_config_files on | off;`
Default	off
Context	location

Enables or disables the `config_files` object, which enumerates the contents of all Angie config files that are currently loaded by the server instance, in the */status/angie/* API section. For example, with this configuration:

```
location /status/ {
    api /status/;
    api_config_files on;
}
```

A query to `/status/angie/` returns approximately this:

```
{
    "version":"1.9.1",
    "address":"192.168.16.5",
    "generation":1,
    "load_time":"2025-05-29T12:58:39.789Z",
    "config_files": {
        "/etc/angie/angie.conf": "...",
        "/etc/angie/mime.types": "..."
    }
}
```

By default, the object is disabled because the config files can contain extra sensitive, confidential details.

Metrics

Angie exposes usage metrics in the `/status/` section of the API; you can make it accessible by defining a respective `location`. Full access:

```
location /status/ {
    api /status/;
}
```

Subtree access, already discussed earlier:

```
location /stats/ {
    api /status/http/server_zones/;
}
```

Example configuration

Configured with `location /status/`, `resolver`, `http upstream`, `http server`, `location`, `cache`, `limit_conn` in `http` and `limit_req` zones:

```
http {

    resolver 127.0.0.53 status_zone=resolver_zone;
    proxy_cache_path /var/cache/angie/cache keys_zone=cache_zone:2m;
    limit_conn_zone $binary_remote_addr zone=limit_conn_zone:10m;
    limit_req_zone $binary_remote_addr zone=limit_req_zone:10m rate=1r/s;
```

```
    upstream upstream {
        zone upstream 256k;
        server backend.example.com service=_example._tcp resolve max_conns=5;
       keepalive 4;
    }

    server {
        server_name www.example.com;
        listen 443 ssl;

        status_zone http_server_zone;
        proxy_cache cache_zone;

        access_log /var/log/access.log main;

        location / {
            root /usr/share/angie/html;
            status_zone location_zone;
            limit_conn limit_conn_zone 1;
            limit_req zone=limit_req_zone burst=5;
        }
        location /status/ {
            api /status/;

            allow 127.0.0.1;
            deny all;
        }

    }
}
```

Responds to `curl https://www.example.com/status/` with the following JSON:

JSON tree

```
{
    "angie": {
        "version":"1.9.1",
        "address":"192.168.16.5",
        "generation":1,
        "load_time":"2025-05-29T12:58:39.789Z"
    },

    "connections": {
        "accepted":2257,
        "dropped":0,
        "active":3,
        "idle":1
    },

    "slabs": {
        "cache_zone": {
            "pages": {
                "used":2,
                "free":506
            },
```

```
            "slots": {
                "64": {
                    "used":1,
                    "free":63,
                    "reqs":1,
                    "fails":0
                },

                "512": {
                    "used":1,
                    "free":7,
                    "reqs":1,
                    "fails":0
                }
            }
        },

        "limit_conn_zone": {
            "pages": {
                "used":2,
                "free":2542
            },

            "slots": {
                "64": {
                    "used":1,
                    "free":63,
                    "reqs":74,
                    "fails":0
                },

                "128": {
                    "used":1,
                    "free":31,
                    "reqs":1,
                    "fails":0
                }
            }
        },

        "limit_req_zone": {
            "pages": {
                "used":2,
                "free":2542
            },

            "slots": {
                "64": {
                    "used":1,
                    "free":63,
                    "reqs":1,
                    "fails":0
                },

                "128": {
                    "used":2,
                    "free":30,
```

```
                    "reqs":3,
                    "fails":0
                }
            }
        }
    },

    "http": {
        "server_zones": {
            "http_server_zone": {
                "ssl": {
                    "handshaked":4174,
                    "reuses":0,
                    "timedout":0,
                    "failed":0
                },

                "requests": {
                    "total":4327,
                    "processing":0,
                    "discarded":8
                },

                "responses": {
                    "200":4305,
                    "302":12,
                    "404":4
                },

                "data": {
                    "received":733955,
                    "sent":59207757
                }
            }
        },

        "location_zones": {
            "location_zone": {
                "requests": {
                    "total":4158,
                    "discarded":0
                },

                "responses": {
                    "200":4157,
                    "304":1
                },

                "data": {
                    "received":538200,
                    "sent":177606236
                }
            }
        },
        "caches": {
            "cache_zone": {
                "size":0,
```

```
            "cold":false,
            "hit": {
                "responses":0,
                "bytes":0
            },

            "stale": {
                "responses":0,
                "bytes":0
            },

            "updating": {
                "responses":0,
                "bytes":0
            },

            "revalidated": {
                "responses":0,
                "bytes":0
            },

            "miss": {
                "responses":0,
                "bytes":0,
                "responses_written":0,
                "bytes_written":0
            },

            "expired": {
                "responses":0,
                "bytes":0,
                "responses_written":0,
                "bytes_written":0
            },

            "bypass": {
                "responses":0,
                "bytes":0,
                "responses_written":0,
                "bytes_written":0
            }
        }
    },

    "limit_conns": {
        "limit_conn_zone": {
            "passed":73,
            "skipped":0,
            "rejected":0,
            "exhausted":0
        }
    },

    "limit_reqs": {
        "limit_req_zone": {
            "passed":54816,
            "skipped":0,
```

```
                "delayed":65,
                "rejected":26,
                "exhausted":0
            }
        },

        "upstreams": {
            "upstream": {
                "peers": {
                    "192.168.16.4:80": {
                        "server":"backend.example.com",
                        "service":"_example._tcp",
                        "backup":false,
                        "weight":5,
                        "state":"up",
                        "selected": {
                            "current":2,
                            "total":232
                        },

                        "max_conns":5,
                        "responses": {
                            "200":222,
                            "302":12
                        },

                        "data": {
                            "sent":543866,
                            "received":27349934
                        },

                        "health": {
                            "fails":0,
                            "unavailable":0,
                            "downtime":0
                        },

                        "sid":"<server_id>"
                    }
                },

                "keepalive":2
            }
        }
    },

    "resolvers": {
        "resolver_zone": {
            "queries": {
                "name":442,
                "srv":2,
                "addr":0
            },

            "responses": {
                "success":440,
                "timedout":1,
```

```
                "format_error":0,
                "server_failure":1,
                "not_found":1,
                "unimplemented":0,
                "refused":1,
                "other":0
            }
        }
    }
}
```


Each JSON branch can be requested separately with the request constructed accordingly, e.g.:

```
$ curl https://www.example.com/status/angie
$ curl https://www.example.com/status/connections
$ curl https://www.example.com/status/slabs
$ curl https://www.example.com/status/slabs/<zone>/slots
$ curl https://www.example.com/status/slabs/<zone>/slots/64
$ curl https://www.example.com/status/http/
$ curl https://www.example.com/status/http/server_zones
$ curl https://www.example.com/status/http/server_zones/<http_server_zone>
$ curl https://www.example.com/status/http/server_zones/<http_server_zone>/ssl
```

Note

By default, the module uses ISO 8601 strings for date values; to use the integer epoch format instead, add the `date=epoch` parameter to the query string:

```
$ curl https://www.example.com/status/angie/load_time

  "2024-04-01T00:59:59+01:00"

$ curl https://www.example.com/status/angie/load_time?date=epoch

  1711929599
```

Server status

`/status/angie`


```
{
    "version": "1.9.1",
    "build_time": "2025-05-29T16:05:43.805Z",
    "address": "192.168.16.5",
    "generation": 1,
    "load_time": "2025-05-29T16:15:43.805Z"
    "config_files": {
        "/etc/angie/angie.conf": "...",
        "/etc/angie/mime.types": "..."
    }
}
```

`version`	String; version of the running Angie web server
`build`	String; particular build name when it specified during compilation
`build_time`	String; the build time of the Angie executable in the *date* format
`address`	String; the address of the server that accepted API request
`generation`	Number; total number of configuration reloads since last start
`load_time`	String or number; time of the last configuration reload, formatted as a *date*; strings have millisecond resolution

`config_files`	Object; its members are absolute pathnames of all Angie configuration files that are currently loaded by the server instance, and their values are string representations of the files' contents, for example:
	`{`
	`"/etc/angie/angie.conf": "server {\n listen 80;\n # ...\ ↪n\n}\n"`
	`}`
	Caution
	The `config_files` object is available in `/status/angie/` only if the *api_config_files* directive is enabled.

Connections global metrics

`/status/connections`

```
{
  "accepted": 2257,
  "dropped": 0,
  "active": 3,
  "idle": 1
}
```

`accepted`	Number; the total number of accepted client connections
`dropped`	Number; the total number of dropped client connections
`active`	Number; the current number of active client connections
`idle`	Number; the current number of idle client connections

Slab allocator metrics of shared memory zones

`/status/slabs/<zone>`

Usage statistics of shared memory zones that utilize slab allocation, such as *limit_conn*, *limit_req*, and *HTTP cache*:

```
limit_conn_zone $binary_remote_addr zone=limit_conn_zone:10m;
limit_req_zone $binary_remote_addr zone=limit_req_zone:10m rate=1r/s;
proxy_cache cache_zone;
```

The specified shared memory zone will collect the following statistics:

`pages`	Object; memory pages statistics
`used`	Number; the number of currently used memory pages
`free`	Number; the number of currently free memory pages
`slots`	Object; memory slots statistics for each slot size. The `slots` object contains fields for requested memory slot sizes (e.g. `8`, `16`, `32`, etc., up to half of the page size)
`used`	Number; the number of currently used memory slots of specified size
`free`	Number; the number of currently free memory slots of specified size
`reqs`	Number; the total number of attempts to allocate memory slot of specified size
`fails`	Number; the number of unsuccessful attempts to allocate memory slot of specified size

Example:

```
{
  "pages": {
    "used": 2,
    "free": 506
  },

  "slots": {
    "64": {
      "used": 1,
      "free": 63,
      "reqs": 1,
      "fails": 0
  }
}
```

Resolver DNS queries

`/status/resolvers/<zone>`

To collect resolver statistics, the *resolver* directive must set the `status_zone` parameter (*HTTP* and *Stream*):

```
resolver 127.0.0.53 status_zone=resolver_zone;
```

The specified shared memory zone will collect the following statistics:

`queries`	Object; queries statistics
`name`	Number; the number of queries to resolve names to addresses (A and AAAA queries)
`srv`	Number; the number of queries to resolve services to addresses (SRV queries)
`addr`	Number; the number of queries to resolve addresses to names (PTR queries)
`responses`	Object; responses statistics
`success`	Number; the number of successful responses
`timedout`	Number; the number of timed out queries
`format_error`	Number; the number responses with code 1 (Format Error)
`server_failure`	Number; the number responses with code 2 (Server Failure)
`not_found`	Number; the number responses with code 3 (Name Error)
`unimplemented`	Number; the number responses with code 4 (Not Implemented)
`refused`	Number; the number responses with code 5 (Refused)
`other`	Number; the number of queries completed with other non-zero code
`sent`	Object; sent DNS queries statistics
`a`	Number; the number of A type queries
`aaaa`	Number; the number of AAAA type queries
`ptr`	Number; the number of PTR type queries
`srv`	Number; the number of SRV type queries

The response codes are described in RFC 1035, section 4.1.1.

Various DNS record types are detailed in RFC 1035, RFC 2782, and RFC 3596.

Example:

```
{
  "queries": {
    "name": 442,
    "srv": 2,
    "addr": 0
  },

  "responses": {
    "success": 440,
    "timedout": 1,
    "format_error": 0,
    "server_failure": 1,
    "not_found": 1,
    "unimplemented": 0,
    "refused": 1,
    "other": 0
  },

  "sent": {
    "a": 185,
    "aaaa": 245,
    "srv": 2,
    "ptr": 12
  }
}
```

HTTP server and location

`/status/http/server_zones/<zone>`

To collect the `server` metrics, set the *status_zone* directive in the *server* context:

```
server {
    ...
    status_zone server_zone;
}
```

To group the metrics by a custom value, use the alternative syntax. Here, the metrics are aggregated by *$host*, with each group reported as a standalone zone:

```
status_zone $host zone=server_zone:5;
```

The specified shared memory zone will collect the following statistics:

`ssl`	Object; SSL statistics. Present if `server` sets `listen ssl;`
`handshaked`	Number; the total number of successful SSL handshakes
`reuses`	Number; the total number of session reuses during SSL handshake
`timedout`	Number; the total number of timed out SSL handshakes
`failed`	Number; the total number of failed SSL handshakes
`requests`	Object; requests statistics
`total`	Number; the total number of client requests
`processing`	Number; the number of currently being processed client requests
`discarded`	Number; the total number of client requests completed without sending a response
`responses`	Object; responses statistics
`<code>`	Number; a non-zero number of responses with status <code> (100-599)
`xxx`	Number; a non-zero number of responses with other status codes
`data`	Object; data statistics
`received`	Number; the total number of bytes received from clients
`sent`	Number; the total number of bytes sent to clients

Example:

```
{
    "ssl":{
        "handshaked":4174,
        "reuses":0,
        "timedout":0,
        "failed":0
    },

    "requests":{
        "total":4327,
        "processing":0,
        "discarded":0
    },

    "responses":{
        "200":4305,
        "302":6,
        "304":12,
        "404":4
    },

    "data":{
        "received":733955,
        "sent":59207757
    }
}
```

`/status/http/location_zones/<zone>`

To collect the `location` metrics, set the *status_zone* directive in the context of *location* or *if in location*:

```
location / {
    root /usr/share/angie/html;
    status_zone location_zone;

    if ($request_uri ~* "^/condition") {
        # ...
        status_zone if_location_zone;
    }
}
```

To group the metrics by a custom value, use the alternative syntax. Here, the metrics are aggregated by *$host*, with each group reported as a standalone zone:

```
status_zone $host zone=server_zone:5;
```

The specified shared memory zone will collect the following statistics:

`requests`	Object; requests statistics
`total`	Number; the total number of client requests
`discarded`	Number; the total number of client requests completed without sending a response
`responses`	Object; responses statistics
`<code>`	Number; a non-zero number of responses with status <code> (100-599)
`xxx`	Number; a non-zero number of responses with other status codes
`data`	Object; data statistics
`received`	Number; the total number of bytes received from clients
`sent`	Number; the total number of bytes sent to clients

Example:

```
{
  "requests": {
    "total": 4158,
    "discarded": 0
  },

  "responses": {
    "200": 4157,
    "304": 1
  },

  "data": {
    "received": 538200,
    "sent": 177606236
  }
}
```

Stream server

`/status/stream/server_zones/<zone>`

To collect the `server` metrics, set the *status_zone* directive in the *server* context:

```
server {
    ...
    status_zone server_zone;
}
```

To group the metrics by a custom value, use the alternative syntax. Here, the metrics are aggregated by *$host*, with each group reported as a standalone zone:

```
status_zone $host zone=server_zone:5;
```

The specified shared memory zone will collect the following statistics:

`ssl`	Object; SSL statistics. Present if `server` sets `listen ssl;`
`handshaked`	Number; the total number of successful SSL handshakes
`reuses`	Number; the total number of session reuses during SSL handshake
`timedout`	Number; the total number of timed out SSL handshakes
`failed`	Number; the total number of failed SSL handshakes
`connections`	Object; connections statistics
`total`	Number; the total number of client connections
`processing`	Number; the number of currently being processed client connections
`discarded`	Number; the total number of client connections completed without creating a session
`passed`	Number; the total number of client connections relayed to another listening port with `pass` directives
`sessions`	Object; sessions statistics
`success`	Number; the number of sessions completed with code 200, which means successful completion
`invalid`	Number; the number of sessions completed with code 400, which happens when client data could not be parsed, e.g. the PROXY protocol header
`forbidden`	Number; the number of sessions completed with code 403, when access was forbidden, for example, when access is limited for certain client addresses
`internal_error`	Number; the number of sessions completed with code 500, the internal server error
`bad_gateway`	Number; the number of sessions completed with code 502, bad gateway, for example, if an upstream server could not be selected or reached
`service_unavailable`	Number; the number of sessions completed with code 503, service unavailable, for example, when access is limited by the number of connections
`data`	Object; data statistics
`received`	Number; the total number of bytes received from clients
`sent`	Number; the total number of bytes sent to clients

Example:

```
{
  "ssl": {
    "handshaked": 24,
    "reuses": 0,
    "timedout": 0,
    "failed": 0
  },

  "connections": {
    "total": 24,
    "processing": 1,
    "discarded": 0,
    "passed": 2
  },
```

```
  "sessions": {
    "success": 24,
    "invalid": 0,
    "forbidden": 0,
    "internal_error": 0,
    "bad_gateway": 0,
    "service_unavailable": 0
  },

  "data": {
    "received": 2762947,
    "sent": 53495723
  }
}
```


HTTP caches

```
proxy_cache cache_zone;
```

/status/http/caches/<cache>

For each zone configured with *proxy_ cache*, the following data is stored:


```
{
  "name_zone": {
    "size": 0,
    "cold": false,
    "hit": {
      "responses": 0,
      "bytes": 0
    },

    "stale": {
      "responses": 0,
      "bytes": 0
    },

    "updating": {
      "responses": 0,
      "bytes": 0
    },

    "revalidated": {
      "responses": 0,
      "bytes": 0
    },

    "miss": {
      "responses": 0,
      "bytes": 0,
      "responses_written": 0,
      "bytes_written": 0
    },

    "expired": {
```

```
      "responses": 0,
      "bytes": 0,
      "responses_written": 0,
      "bytes_written": 0
    },

    "bypass": {
      "responses": 0,
      "bytes": 0,
      "responses_written": 0,
      "bytes_written": 0
    }
  }
}
```


`size`	Number; the current size of the cache
`max_size`	Number; configured limit on the maximum size of the cache
`cold`	Boolean; `true` while the *cache loader* loads data from disk
`hit`	Object; statistics of valid cached responses (*proxy_cache_valid*)
`responses`	Number; the total number of responses read from the cache
`bytes`	Number; the total number of bytes read from the cache
`stale`	Object; statistics of expired responses taken from the cache (*proxy_cache_use_stale*)
`responses`	Number; the total number of responses read from the cache
`bytes`	Number; the total number of bytes read from the cache
`updating`	Object; statistics of expired responses taken from the cache while responses were being updated (*proxy_cache_use_stale* updating)
`responses`	Number; the total number of responses read from the cache
`bytes`	Number; the total number of bytes read from the cache
`revalidated`	Object; statistics of expired and revalidated responses taken from the cache (*proxy_cache_revalidate*)
`responses`	Number; the total number of responses read from the cache
`bytes`	Number; the total number of bytes read from the cache
`miss`	Object; statistics of responses not found in the cache
`responses`	Number; the total number of corresponding responses
`bytes`	Number; the total number of bytes read from the proxied server
`responses_written`	Number; the total number of responses written to the cache
`bytes_written`	Number; the total number of bytes written to the cache
`expired`	Object; statistics of expired responses not taken from the cache
`responses`	Number; the total number of corresponding responses
`bytes`	Number; the total number of bytes read from the proxied server
`responses_written`	Number; the total number of responses written to the cache
`bytes_written`	Number; the total number of bytes written to the cache
`bypass`	Object; statistics of responses not looked up in the cache (*proxy_cache_bypass*)
`responses`	Number; the total number of corresponding responses
`bytes`	Number; the total number of bytes read from the proxied server
`responses_written`	Number; the total number of responses written to the cache
`bytes_written`	Number; the total number of bytes written to the cache

Added in version 1.2.0: PRO

In Angie PRO, if *cache sharding* is enabled with *proxy_cache_path* directives, individual shards are exposed as object members of a `shards` object:

`shards`	Object; lists individual shards as members
`<shard>`	Object; represents an individual shard with its cache path for name
`size`	Number; the shard's current size
`max_size`	Number; maximum shard size, if configured
`cold`	Boolean; `true` while the *cache loader* loads data from disk

```
{
  "name_zone": {
    "shards": {
        "/path/to/shard1": {
            "size": 0,
            "cold": false
        },

        "/path/to/shard2": {
            "size": 0,
            "cold": false
        }
    }
}
```

limit_conn

```
limit_conn_zone $binary_remote_addr zone=limit_conn_zone:10m;
```

`/status/http/limit_conns/<zone>`, `/status/stream/limit_conns/<zone>`

Objects for each configured *limit_conn in http* or *limit_conn in stream* contexts with the following fields:

```
{
  "passed": 73,
  "skipped": 0,
  "rejected": 0,
  "exhausted": 0
}
```

`passed`	Number; the total number of passed connections
`skipped`	Number; the total number of connections passed with zero-length key, or key exceeding 255 bytes
`rejected`	Number; the total number of connections exceeding the configured limit
`exhausted`	Number; the total number of connections rejected due to exhaustion of zone storage

limit_req

```
limit_req_zone $binary_remote_addr zone=limit_req_zone:10m rate=1r/s;
```

`/status/http/limit_reqs/<zone>`

Objects for each configured *limit_req* with the following fields:

```
{
  "passed": 54816,
```

```
  "skipped": 0,
  "delayed": 65,
  "rejected": 26,
  "exhausted": 0
}
```


`passed`	Number; the total number of passed connections
`skipped`	Number; the total number of requests passed with zero-length key, or key exceeding 65535 bytes
`delayed`	Number; the total number of delayed requests
`rejected`	Number; the total number of rejected requests
`exhausted`	Number; the total number of requests rejected due to exhaustion of zone storage

HTTP upstream

Added in version 1.1.0.

To enable collection of the following metrics, set the *zone* directive in the *upstream* context, for instance:

```
upstream upstream {
    zone upstream 256k;
    server backend.example.com service=_example._tcp resolve max_conns=5;
    keepalive 4;
}
```

`/status/http/upstreams/<upstream>`

where *<upstream>* is the name of any *upstream* specified with the *zone* directive


```
{
    "peers": {
        "192.168.16.4:80": {
            "server": "backend.example.com",
            "service": "_example._tcp",
            "backup": false,
            "weight": 5,
            "state": "up",
            "selected": {
                "current": 2,
                "total": 232
            },

            "max_conns": 5,
            "responses": {
                "200": 222,
                "302": 12
            },

            "data": {
                "sent": 543866,
                "received": 27349934
            },

            "health": {
                "fails": 0,
```

```
                "unavailable": 0,
                "downtime": 0
            },

            "sid": "<server_id>"
        }
    },

    "keepalive": 2
}
```

`peers`	Object; contains the metrics of the upstream's peers as subobjects whose names are canonical representations of the peers' addresses. Members of each subobject:
`server`	String; the parameter of the *server* directive
`service`	String; name of service as it's specified in *server* directive, if configured
`slow_start` (PRO 1.4.0+)	Number; the specified *slow_start* value for the server, expressed in seconds. When setting the value via the *respective subsection* of the dynamic configuration API, you can specify either a number or a *time* value with millisecond precision.
`backup`	Boolean; `true` for backup servers
`weight`	Number; configured *weight*
`state`	String; the current state of the peer and what requests are sent to it: • `busy`: indicates that the number of requests to the server has reached the limit set by *max_conns*, and no new requests are sent • `down`: manually disabled, no requests are sent • `recovering`: recovering after a failure according to *slow_start*, more and more requests are sent over time • `unavailable`: reached the *max_fails* limit, only trial client requests are sent at intervals defined by *fail_timeout*; • `up`: operational, requests are sent as usual Additional states in Angie PRO: • `checking`: configured as `essential` and being checked, only *probe requests* are sent • `draining`: similar to `down`, but requests from previously bound sessions (via *sticky*) are still sent • `unhealthy`: non-operational, only *probe requests* are sent
`selected`	Object; peer selection statistics
`current`	Number; the current number of connections to peer
`total`	Number; total number of requests forwarded to peer
`last`	String or number; time when peer was last selected, formatted as a *date*
`max_conns`	Number; the configured *maximum* number of simultaneous connections, if specified
`responses`	Object; responses statistics
`<code>`	Number; a non-zero number of responses with status <code> (100-599)
`xxx`	Number; a non-zero number of responses with other status codes
`data`	Object; data statistics
`received`	Number; the total number of bytes received from peer
`sent`	Number; the total number of bytes sent to peer
`health`	Object; health statistics
`fails`	Number; the total number of unsuccessful attempts to communicate with the peer
`unavailable`	Number; how many times peer became `unavailable` due to reaching the *max_fails* limit
`downtime`	Number; the total time (in milliseconds) when peer was `unavailable` for selection
`downstart`	String or number; time when peer became `unavailable`, formatted as a *date*
`header_time` (PRO 1.3.0+)	Number; average time (in milliseconds) to receive the response headers from the peer; see *response_time_factor (PRO)*
`response_time` (PRO 1.3.0+)	Number; average time (in milliseconds) to receive the entire peer response; see *response_time_factor (PRO)*
`sid`	String; *configured id* of the server in upstream group
`keepalive`	Number; the number of currently cached connections
`backup_switch`	Object; contains the current state of the active backup logic, present if *backup_switch (PRO)* is configured for the upstream
`active`	Number; active group identifier, if any
`timeout`	Number; time to expire in milliseconds, after which the balancer will recheck the groups for healthy peers; does not appear for the primary group

`health/probes` **(PRO)**

Changed in version 1.2.0: PRO

If the upstream has *upstream_probe (PRO)* probes configured, the `health` object also has a `probes` subobject that stores the peer's health probe counters, while the peer's `state` can also be `checking` and `unhealthy`, apart from the values listed in the table above:

```
{
    "192.168.16.4:80": {
        "state": "unhealthy",
        "...": "...",
        "health": {
            "...": "...",
            "probes": {
                "count": 10,
                "fails": 10,
                "last": "2025-05-29T09:56:07Z"
            }
        }
    }
}
```

The `checking` value of `state` isn't counted as `downtime` and means that the peer, which has a probe configured as `essential`, hasn't been checked yet; the `unhealthy` value means that the peer is malfunctioning. Both states also imply that the peer isn't included in load balancing. For details of health probes, see *upstream_probe*.

Counters in `probes`:

`count`	Number; total probes for this peer
`fails`	Number; total failed probes
`last`	String or number; last probe time, formatted as a *date*

`queue`

Changed in version 1.4.0.

If a *request queue* is configured for the upstream, the upstream object also contains a nested `queue` object, which holds counters for requests in the queue:

```
{
    "queue": {
        "queued": 20112,
        "waiting": 1011,
        "dropped": 6031,
        "timedout": 560,
        "overflows": 13
    }
}
```

The counter values are aggregated across all worker processes:

`queued`	Number; total count of requests that entered the queue
`waiting`	Number; current count of requests in the queue
`dropped`	Number; total count of requests removed from the queue due to the client prematurely closing the connection
`timedout`	Number; total count of requests removed from the queue due to timeout
`overflows`	Number; total count of queue overflow occurrences

Stream upstream

To enable collection of the following metrics, set the *zone* directive in the *upstream* context, for instance:

```
upstream upstream {
    zone upstream 256k;
    server backend.example.com service=_example._tcp resolve max_conns=5;
    keepalive 4;
}
```

`/status/stream/upstreams/<upstream>`

Here, *<upstream>* is the name of an *upstream* that is configured with a *zone* directive.

```
{
    "peers": {
        "192.168.16.4:1935": {
            "server": "backend.example.com",
            "service": "_example._tcp",
            "backup": false,
            "weight": 5,
            "state": "up",
            "selected": {
                "current": 2,
                "total": 232
            },

            "max_conns": 5,
            "data": {
                "sent": 543866,
                "received": 27349934
            },

            "health": {
                "fails": 0,
                "unavailable": 0,
                "downtime": 0
            }
        }
    }
}
```

`peers`	Object; contains the metrics of the upstream's peers as subobjects whose names are canonical representations of the peers' addresses. Members of each subobject:
`server`	String; address set by the *server* directive
`service`	String; service name, if set by *server* directive
`slow_start` (PRO 1.4.0+)	Number; the specified *slow_start* value for the server, expressed in seconds. When setting the value via the *respective subsection* of the dynamic configuration API, you can specify either a number or a *time* value with millisecond precision.
`backup`	Boolean; `true` for backup server
`weight`	Number; the *weight* of the peer
`state`	String; the current state of the peer and what requests are sent to it: • `busy`: indicates that the number of requests to the server has reached the limit set by *max_conns*, and no new requests are sent • `down`: manually disabled, no requests are sent • `recovering`: recovering after a failure according to *slow_start*, more and more requests are sent over time • `unavailable`: reached the *max_fails* limit, only trial client requests are sent at intervals defined by *fail_timeout*; • `up`: operational, requests are sent as usual Additional states in Angie PRO: • `checking`: configured as `essential` and being checked, only *probe requests* are sent • `draining`: similar to `down`, but requests from previously bound sessions (via *sticky*) are still sent • `unhealthy`: non-operational, only *probe requests* are sent
`selected`	Object; the peer's selection metrics
`current`	Number; current connections to the peer
`total`	Number; total connections forwarded to the peer
`last`	String or number; time when the peer was last selected, formatted as a *date*
`max_conns`	Number; *maximum* number of simultaneous active connections to the peer, if set
`data`	Object; data transfer metrics
`received`	Number; total bytes received from the peer
`sent`	Number; total bytes sent to the peer
`health`	Object; peer health metrics
`fails`	Number; total failed attempts to reach the peer
`unavailable`	Number; times the peer became `unavailable` due to reaching the *max_fails*
`downtime`	Number; total time (in milliseconds) that the peer was `unavailable` for selection
`downstart`	String or number; time when the peer last became `unavailable`, formatted as a *date*
`connect_time` (PRO 1.4.0+)	Number; average time (in milliseconds) taken to establish a connection with the peer; see the *response_time_factor (PRO)* directive.
`first_byte_time` (PRO 1.4.0+)	Number; average time (in milliseconds) to receive the first byte of the response from the peer; see the *response_time_factor (PRO)* directive.
`last_byte_time` (PRO 1.4.0+)	Number; average time (in milliseconds) to receive the complete response from the peer; see the *response_time_factor (PRO)* directive.

Changed in version 1.4.0: PRO

In Angie PRO, if the upstream has *upstream_probe (PRO)* probes configured, the `health` object also has a `probes` subobject that stores the peer's health probe counters, while the peer's `state` can also be `checking` and `unhealthy`, apart from the values listed in the table above:

```
{
    "192.168.16.4:80": {
        "state": "unhealthy",
        "...": "...",
        "health": {
            "...": "...",
            "probes": {
                "count": 2,
                "fails": 2,
                "last": "2025-05-29T11:03:54Z"
            }
        }
    }
}
```

The `checking` value of `state` isn't counted as `downtime` and means that the peer, which has a probe configured as `essential`, hasn't been checked yet; the `unhealthy` value means that the peer is malfunctioning. Both states also imply that the peer isn't included in load balancing. For details of health probes, see *upstream_probe*.

Counters in `probes`:

`count`	Number; total probes for this peer
`fails`	Number; total failed probes
`last`	String or number; last probe time, formatted as a *date*

Dynamic Configuration API (PRO only)

Added in version 1.2.0.

The API includes a `/config` section that enables dynamic updates to Angie's configuration in JSON with `PUT`, `PATCH`, and `DELETE` HTTP requests. All updates are atomic; new settings are applied as a whole, or none are applied at all. On error, Angie reports the reason.

Subsections of `/config`

Currently, configuration of individual servers within upstreams is available in the `/config` section for the *HTTP* and *stream* modules; the number of settings eligible for dynamic configuration is steadily increasing.

`/config/http/upstreams/<upstream>/servers/<name>`

Enables configuring individual upstream peers, including deleting existing peers or adding new ones.

URI path parameters:

`<upstream>`	Name of the upstream; to be configurable via `/config`, it must have a *zone* directive configured, defining a shared memory zone.
`<name>`	The peer's name within the upstream, defined as `<service>@<host>`, where: • `<service>@` is an optional service name, used for SRV record resolution. • `<host>` is the domain name of the service (if `resolve` is present) or its IP; an optional port can be defined here.

For example, the following configuration:

```
upstream backend {
    server backend.example.com service=_http._tcp resolve;
    server 127.0.0.1;
    zone backend 1m;
}
```

Allows the following peer names:

```
$ curl http://127.0.0.1/config/http/upstreams/backend/servers/_http._tcp@backend.
↪example.com/
$ curl http://127.0.0.1/config/http/upstreams/backend/servers/127.0.0.1:80/
```

This API subsection enables setting the `weight`, `max_conns`, `max_fails`, `fail_timeout`, `backup`, `down` and `sid` parameters, as described in *server*.

Note

There is no separate `drain` option here; to enable `drain`, set `down` to the string value `drain`:

```
$ curl -X PUT -d \"drain\" \
  http://127.0.0.1/config/http/upstreams/backend/servers/backend.example.com/down
```

Example:

```
curl http://127.0.0.1/config/http/upstreams/backend/servers/backend.example.com?
↪defaults=on
```

```
{
    "weight": 1,
    "max_conns": 0,
    "max_fails": 1,
    "fail_timeout": 10,
    "backup": true,
    "down": false,
    "sid": ""
}
```

Actually available parameters are limited to the ones supported by the current load balancing method of the *upstream*. So, if the upstream is configured as `random`:

```
upstream backend {
    zone backend 256k;
    server backend.example.com resolve max_conns=5;
    random;
}
```

You will be unable to add a new peer that defines `backup`:

```
$ curl -X PUT -d '{ "backup": true }' \
    http://127.0.0.1/config/http/upstreams/backend/servers/backend1.example.com
```

```
{
    "error": "FormatError",
    "description": "The \"backup\" field is unknown."
}
```

Note

Even with a compatible load balancing method, the `backup` parameter can only be set at new peer creation.

`/config/stream/upstreams/<upstream>/servers/<name>`

Enables configuring individual upstream peers, including deleting existing peers or adding new ones.

URI path parameters:

`<upstream>`	Name of the upstream; to be configurable via `/config`, it must have a *zone* directive configured, defining a shared memory zone.
`<name>`	The peer's name within the upstream, defined as `<service>@<host>`, where: • `<service>@` is an optional service name, used for SRV record resolution. • `<host>` is the domain name of the service (if `resolve` is present) or its IP; an optional port can be defined here.

For example, the following configuration:

```
upstream backend {
    server backend.example.com:8080 service=_example._tcp resolve;
    server 127.0.0.1:12345;
    zone backend 1m;
}
```

Allows the following peer names:

```
$ curl http://127.0.0.1/config/stream/upstreams/backend/servers/_example._tcp@backend.
↪example.com:8080/
$ curl http://127.0.0.1/config/stream/upstreams/backend/servers/127.0.0.1:12345/
```

This API subsection enables setting the `weight`, `max_conns`, `max_fails`, `fail_timeout`, `backup` and `down` parameters, as described in *server*.

Note

There is no separate `drain` option here; to enable `drain`, set `down` to the string value `drain`:

```
$ curl -X PUT -d \"drain\" \
  http://127.0.0.1/config/stream/upstreams/backend/servers/backend.example.com/down
```

Example:

```
curl http://127.0.0.1/config/stream/upstreams/backend/servers/backend.example.com?
↪defaults=on
```

```
{
    "weight": 1,
    "max_conns": 0,
    "max_fails": 1,
    "fail_timeout": 10,
    "backup": true,
```

```
    "down": false,
}
```

Actually available parameters are limited to the ones supported by the current load balancing method of the *upstream*. So, if the upstream is configured as `random`:

```
upstream backend {
    zone backend 256k;
    server backend.example.com resolve max_conns=5;
    random;
}
```

You will be unable to add a new peer that defines `backup`:

```
$ curl -X PUT -d '{ "backup": true }' \
    http://127.0.0.1/config/stream/upstreams/backend/servers/backend1.example.com
```

```
{
    "error": "FormatError",
    "description": "The \"backup\" field is unknown."
}
```

Note

Even with a compatible load balancing method, the `backup` parameter can only be set at new peer creation.

When deleting servers, you can set the `connection_drop=<value>` argument (PRO) to override the *proxy_ connection_ drop* settings:

```
$ curl -X DELETE \
    http://127.0.0.1/config/stream/upstreams/backend/servers/backend1.example.com?
↪connection_drop=off

$ curl -X DELETE \
    http://127.0.0.1/config/stream/upstreams/backend/servers/backend2.example.com?
↪connection_drop=on

$ curl -X DELETE \
    http://127.0.0.1/config/stream/upstreams/backend/servers/backend3.example.com?
↪connection_drop=1000
```

HTTP Methods

Let's consider the semantics of all HTTP methods applicable to this section, given this upstream configuration:

```
http {
    # ...

    upstream backend {
        zone upstream 256k;
        server backend.example.com resolve max_conns=5;
        # ...
    }
```

```
    server {
        # ...

        location /config/ {
            api /config/;

            allow 127.0.0.1;
            deny all;
        }
    }
}
```

GET

The `GET` HTTP method queries an entity at any existing path within `/config`, just as it does for other API sections.

For example, the `/config/http/upstreams/backend/servers/` upstream server branch enables these queries:

```
$ curl http://127.0.0.1/config/http/upstreams/backend/servers/backend.example.com/max_
↪conns
$ curl http://127.0.0.1/config/http/upstreams/backend/servers/backend.example.com
$ curl http://127.0.0.1/config/http/upstreams/backend/servers
$ # ...
$ curl http://127.0.0.1/config
```

You can obtain default parameter values with `defaults=on`:

```
$ curl http://127.0.0.1/config/http/upstreams/backend/servers?defaults=on
```

```
{
    "backend.example.com": {
        "weight": 1,
        "max_conns": 5,
        "max_fails": 1,
        "fail_timeout": 10,
        "backup": false,
        "down": false,
        "sid": ""
    }
}
```

PUT

The `PUT` HTTP method creates a new JSON entity at the specified path or *entirely* replaces an existing one.

For example, to set the `max_fails` parameter, not specified earlier, of the `backend.example.com` server within the `backend` upstream:

```
$ curl -X PUT -d '2' \
    http://127.0.0.1/config/http/upstreams/backend/servers/backend.example.com/max_
↪fails
```

```
{
    "success": "Updated",
```

```
    "description": "Existing configuration API entity \"/config/http/upstreams/
↪backend/servers/backend.example.com/max_fails\" was updated with replacing."
}
```

Verify the changes:

```
$ curl http://127.0.0.1/config/http/upstreams/backend/servers/backend.example.com
```

```
{
    "max_conns": 5,
    "max_fails": 2
}
```

DELETE

The `DELETE` HTTP method deletes *previously defined* settings at the specified path; at doing that, it returns to the default values if there are any.

For example, to delete the previously set `max_fails` parameter of the `backend.example.com` server within the `backend` upstream:

```
$ curl -X DELETE \
    http://127.0.0.1/config/http/upstreams/backend/servers/backend.example.com/max_
↪fails
```

```
{
    "success": "Reset",
    "description": "Configuration API entity \"/config/http/upstreams/backend/servers/
↪backend.example.com/max_fails\" was reset to default."
}
```

Verify the changes using `defaults=on`:

```
$ curl http://127.0.0.1/config/http/upstreams/backend/servers/backend.example.com?
↪defaults=on
```

```
{
    "weight": 1,
    "max_conns": 5,
    "max_fails": 1,
    "fail_timeout": 10,
    "backup": false,
    "down": false,
    "sid": ""
}
```

The `max_fails` setting is back to its default value.

When deleting servers, you can set the `connection_drop=<value>` argument (PRO) to override the *proxy_connection_drop*, *grpc_connection_drop*, *fastcgi_connection_drop*, *scgi_connection_drop*, and *uwsgi_connection_drop* settings:

```
$ curl -X DELETE \
    http://127.0.0.1/config/http/upstreams/backend/servers/backend1.example.com?
↪connection_drop=off

$ curl -X DELETE \
    http://127.0.0.1/config/http/upstreams/backend/servers/backend2.example.com?
```

```
↪connection_drop=on

$ curl -X DELETE \
    http://127.0.0.1/config/http/upstreams/backend/servers/backend3.example.com?
↪connection_drop=1000
```

PATCH

The `PATCH` HTTP method creates a new entity at the specified path or partially replaces or complements an existing one (RFC 7386) by supplying a JSON definition in its payload.

The method operates as follows: if the entities from the new definition exist in the configuration, they are overwritten; otherwise, they are added.

For example, to change the `down` setting of the `backend.example.com` server within the `backend` upstream, leaving the rest intact:

```
$ curl -X PATCH -d '{ "down": true }' \
    http://127.0.0.1/config/http/upstreams/backend/servers/backend.example.com
```

```
{
    "success": "Updated",
    "description": "Existing configuration API entity \"/config/http/upstreams/
↪backend/servers/backend.example.com\" was updated with merging."
}
```

Verify the changes:

```
$ curl http://127.0.0.1/config/http/upstreams/backend/servers/backend.example.com
```

```
{
    "max_conns": 5,
    "down": true
}
```

The JSON object supplied with the `PATCH` request *was merged* with the existing one instead of overwriting it, as would be the case with `PUT`.

The `null` values are a corner case; they are used to *delete* specific configuration items during such merge.

> **Note**
>
> This deletion is identical to `DELETE`; in particular, it reinstates the default values.

For example, to delete the `down` setting added earlier and simultaneously update `max_conns`:

```
$ curl -X PATCH -d '{ "down": null, "max_conns": 6 }' \
    http://127.0.0.1/config/http/upstreams/backend/servers/backend.example.com
```

```
{
    "success": "Updated",
    "description": "Existing configuration API entity \"/config/http/upstreams/
↪backend/servers/backend.example.com\" was updated with merging."
}
```

Verify the changes:

```
$ curl http://127.0.0.1/config/http/upstreams/backend/servers/backend.example.com
```

```
{
    "max_conns": 6
}
```

The `down` parameter, for which a `null` was supplied, was deleted; `max_conns` was updated.

Auth Basic

Allows limiting access to resources by validating the user name and password using the "HTTP Basic Authentication" protocol.

Access can also be limited by *address* or by the *result of subrequest*. Simultaneous limitation of access by address and by password is controlled by the *satisfy* directive.

Configuration Example

```
location / {
    auth_basic           "closed site";
    auth_basic_user_file conf/htpasswd;
}
```

Directives

auth_basic

Syntax	`auth_basic` *string* | `off`;
Default	`auth_basic off;`
Context	http, server, location, limit_except

Enables validation of user name and password using the "HTTP Basic Authentication" protocol. The specified parameter is used as a *realm*. Parameter value can contain variables.

`off`	cancels the effect of the *auth_basic* directive inherited from the previous configuration level

auth_basic_user_file

Syntax	`auth_basic_user_file` *file*;
Default	—
Context	http, server, location, limit_except

Specifies a `file` that keeps user names and passwords, in the following format:

```
# comment
name1:password1
name2:password2:comment
name3:password3
```

The `file` name can contain variables.

The following password types are supported:

- encrypted with the *crypt()* function; can be generated using the `htpasswd` utility from the Apache HTTP Server distribution or the "*openssl passwd*" command;
- hashed with the Apache variant of the MD5-based password algorithm (apr1); can be generated with the same tools;
- specified by the "*{scheme}data*" syntax as described in RFC 2307; currently implemented schemes include *PLAIN* (an example one, should not be used), *SHA* (plain SHA-1 hashing, should not be used) and *SSHA* (salted SHA-1 hashing, used by some software packages, notably OpenLDAP and Dovecot).

Caution

Support for SHA scheme was added only to aid in migration from other web servers. It should not be used for new passwords, since unsalted SHA-1 hashing that it employs is vulnerable to rainbow table attacks.

Auth Request

Implements client authorization based on the result of a subrequest. If the subrequest returns a 2xx response code, the access is allowed. If it returns 401 or 403, the access is denied with the corresponding error code. Any other response code returned by the subrequest is considered an error.

For the 401 error, the client also receives the "WWW-Authenticate" header from the subrequest response.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-http_auth_request_module` build option.

In packages and images from our repos, the module is included in the build.

The module may be combined with other access modules, such as *Access* and *Auth Basic*, via the *satisfy* directive.

Configuration Example

```
location /private/ {
    auth_request /auth;
#    ...
}

location = /auth {
    proxy_pass ...
    proxy_pass_request_body off;
    proxy_set_header Content-Length "";
    proxy_set_header X-Original-URI $request_uri;
}
```

Directives

auth_request

Syntax	`auth_request uri | off;`
Default	`auth_request off;`
Context	http, server, location

Enables authorization based on the result of a subrequest and sets the URI to which the subrequest will be sent.

auth_request_set

Syntax	`auth_request_set` *$variable value*;
Default	—
Context	http, server, location

Sets the request variable to the given value after the authorization request completes. The value may contain variables from the authorization request, such as `$upstream_http_*`.

AutoIndex

The module processes requests ending with the slash character (`/`) and produces a directory listing. Usually, a request is passed to the *http_autoindex* module when the *http_index* module cannot find an index file.

Configuration Example

```
location / {
    autoindex on;
}
```

Directives

autoindex

Syntax	`autoindex on | off;`
Default	`autoindex off;`
Context	http, server, location

Enables or disables the directory listing output.

autoindex_exact_size

Syntax	`autoindex_exact_size on | off;`
Default	`autoindex_exact_size on;`
Context	http, server, location

For the HTML *format*, specifies whether exact file sizes should be output in the directory listing, or rather rounded to kilobytes, megabytes, and gigabytes.

autoindex_format

Syntax	`autoindex_format html | xml | json | jsonp;`
Default	`autoindex_format html;`
Context	http, server, location

Sets the format of a directory listing.

When the JSONP format is used, the name of a callback function is set with the *callback* request argument. If the argument is missing or has an empty value, then the JSON format is used.

The XML output can be transformed using the *http_xslt* module.

autoindex_localtime

Syntax	`autoindex_localtime on | off;`
Default	`autoindex_localtime off;`
Context	http, server, location

For the HTML *format*, specifies whether times in the directory listing should be output in the local time zone or UTC.

Browser

The module creates variables whose values depend on the value of the "User-Agent" request header field.

Variables

`$modern_browser`

equals the value set by the *modern_browser_value* directive, if a browser was identified as modern;

`$ancient_browser`

equals the value set by the *ancient_browser_value* directive, if a browser was identified as ancient;

`$msie`

equals "1" if a browser was identified as MSIE of any version.

Configuration Example

Choosing an index file:

```
modern_browser_value "modern.";

modern_browser msie      5.5;
modern_browser gecko     1.0.0;
modern_browser opera     9.0;
modern_browser safari    413;
modern_browser konqueror 3.0;

index index.${modern_browser}html index.html;
```

Redirection for old browsers:

```
modern_browser msie      5.0;
modern_browser gecko     0.9.1;
modern_browser opera     8.0;
modern_browser safari    413;
modern_browser konqueror 3.0;

modern_browser unlisted;

ancient_browser Links Lynx netscape4;

if ($ancient_browser) {
    rewrite ^ /ancient.html;
}
```

Directives

ancient_browser

Syntax	`ancient_browser` *string* ...;
Default	—
Context	http, server, location

If any of the specified substrings is found in the "User-Agent" request header field, the browser will be considered ancient. The special string "netscape4" corresponds to the regular expression "^Mozilla/[1-4]".

ancient_browser_value

Syntax	`ancient_browser_value` *string*;
Default	`ancient_browser_value 1;`
Context	http, server, location

Sets a value for the *$ancient_browser* variables.

modern_browser

Syntax	`modern_browser` *browser version*; `modern_browser unlisted`;
Default	—
Context	http, server, location

Specifies a version starting from which a browser is considered modern. A browser can be any one of the following: `msie`, `gecko` (browsers based on Mozilla), `opera`, `safari`, or `konqueror`.

Versions can be specified in the following formats: X, X.X, X.X.X, or X.X.X.X. The maximum values for each of the format are 4000, 4000.99, 4000.99.99, and 4000.99.99.99, respectively.

The special value `unlisted` specifies to consider a browser as modern if it was not listed by the *modern_browser* and *ancient_browser* directives. Otherwise such a browser is considered ancient. If a request does not provide the "User-Agent" field in the header, the browser is treated as not being listed.

modern_browser_value

Syntax	`modern_browser_value` *string*;
Default	`modern_browser_value 1;`
Context	http, server, location

Sets a value for the *$modern_browser* variables.

Charset

The module adds the specified charset to the "Content-Type" response header field. In addition, the module can convert data from one charset to another, with some limitations:

- conversion is performed one way — from server to client,
- only single-byte charsets can be converted
- or single-byte charsets to/from UTF-8.

Configuration Example

```
include        conf/koi-win;

charset        windows-1251;
source_charset koi8-r;
```

Directives

charset

Syntax	`charset` *charset* | `off`;
Default	`charset off;`
Context	http, server, location, if in location

Adds the specified charset to the "Content-Type" response header field. If this charset is different from the charset specified in the *source_charset* directive, a conversion is performed.

The parameter `off` cancels the addition of charset to the "Content-Type" response header field.

A charset can be defined with a variable:

```
charset $charset;
```

In such a case, all possible values of a variable need to be present in the configuration at least once in the form of the *charset_map*, *charset*, or *source_charset* directives. For `utf-8`, `windows-1251`, and `koi8-r` charsets, it is sufficient to include the files `conf/koi-win`, `conf/koi-utf`, and `conf/win-utf` into configuration. For other charsets, simply making a fictitious conversion table works, for example:

```
charset_map iso-8859-5 _ { }
```

In addition, a charset can be set in the "X-Accel-Charset" response header field. This capability can be disabled using the *proxy_ignore_headers*, *fastcgi_ignore_headers*, *uwsgi_ignore_headers*, *scgi_ignore_headers*, and *grpc_ignore_headers* directives.

charset_map

Syntax	`charset_map` *charset1 charset2* { ... }
Default	—
Context	http

Describes the conversion table from one charset to another. A reverse conversion table is built using the same data. Character codes are given in hexadecimal. Missing characters in the range 80-FF are replaced with "*?*". When converting from UTF-8, characters missing in a one-byte charset are replaced with "*&#XXXX;*".

Example:

```
charset_map koi8-r windows-1251 {
    C0 FE ; # small yu
    C1 E0 ; # small a
    C2 E1 ; # small b
    C3 F6 ; # small ts
}
```

When describing a conversion table to UTF-8, codes for the UTF-8 charset should be given in the second column, for example:

```
charset_map koi8-r utf-8 {
    C0 D18E ; # small yu
    C1 D0B0 ; # small a
    C2 D0B1 ; # small b
    C3 D186 ; # small ts
}
```

Full conversion tables from `koi8-r` to `windows-1251`, and from `koi8-r` and `windows-1251` to `utf-8` are provided in the distribution files `conf/koi-win`, `conf/koi-utf`, and `conf/win-utf`.

charset_types

Syntax	`charset_types` mime-*type* ...;
Default	`charset_types text/html text/xml text/plain text/vnd.wap.wml application/javascript application/rss+xml;`
Context	http, server, location

Enables module processing in responses with the specified MIME types in addition to `text/html`. The special value `*` matches any MIME type.

override_charset

Syntax	`override_charset on` | `off`;
Default	`override_charset off;`
Context	http, server, location, if in location

Determines whether a conversion should be performed for answers received from a proxied or a FastCGI/uwsgi/SCGI/gRPC server when the answers already carry a charset in the "Content-Type" response header field. If conversion is enabled, a charset specified in the received response is used as a source charset.

Note

If a response is received in a subrequest then the conversion from the response charset to the main request charset is always performed, regardless of the *override_charset* directive setting.

source_charset

Syntax	`source_charset` *encoding*;
Default	—
Context	http, server, location, if in location

Defines the source charset of a response. If this charset is different from the charset specified in the *charset* directive, a conversion is performed.

DAV

The module is intended for file management automation via the WebDAV protocol. The module processes HTTP and WebDAV methods PUT, DELETE, MKCOL, COPY, and MOVE.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-http_dav_module` build option.

In packages and images from our repos, the module is included in the build.

Important

WebDAV clients that require additional WebDAV methods to operate will not work with this module.

Configuration Example

```
location / {
    root                  /data/www;

    client_body_temp_path /data/client_temp;

    dav_methods PUT DELETE MKCOL COPY MOVE;

    create_full_put_path  on;
    dav_access            group:rw  all:r;

    limit_except GET {
        allow 192.168.1.0/32;
        deny  all;
    }
}
```

Directives

create_full_put_path

Syntax	`create_full_put_path on | off;`
Default	`create_full_put_path off;`
Context	http, server, location

The WebDAV specification only allows creating files in already existing directories. This directive allows creating all needed intermediate directories.

dav_access

Syntax	`dav_access` *users:permissions ...*;
Default	`dav_access user:rw;`
Context	http, server, location

Sets access permissions for newly created files and directories, e.g.:

```
dav_access user:rw group:rw all:r;
```

If any group or all access permissions are specified then user permissions may be omitted:

```
dav_access group:rw all:r;
```

dav_methods

Syntax	`dav_methods off` | *method* ...;
Default	`dav_methods off;`
Context	http, server, location

Allows the specified HTTP and WebDAV methods. The parameter off denies all methods processed by this module. The following methods are supported: PUT, DELETE, MKCOL, COPY, and MOVE.

A file uploaded with the PUT method is first written to a temporary file, and then the file is renamed. Starting from version 0.8.9, temporary files and the persistent store can be put on different file systems. However, be aware that in this case a file is copied across two file systems instead of the cheap renaming operation. It is thus recommended that for any given `location` both saved files and a directory holding temporary files, set by the *client_body_temp_path* directive, are put on the same file system.

When creating a file with the PUT method, it is possible to specify the modification date by passing it in the "Date" header field.

min_delete_depth

Syntax	`min_delete_depth` *number*;
Default	`min_delete_depth 0;`
Context	http, server, location

Allows the DELETE method to remove files provided that the number of elements in a request path is not less than the specified number. For example, the directive

```
min_delete_depth 4;
```

allows removing files on requests

```
/users/00/00/name
/users/00/00/name/pic.jpg
/users/00/00/page.html
```

and denies the removal of

```
/users/00/00
```

Empty GIF

The module emits a single-pixel transparent GIF.

Configuration Example

```
location = /_.gif {
    empty_gif;
}
```

Directives

empty_gif

Syntax	`empty_gif;`
Default	—
Context	location

Turns on module processing in the surrounding `location`.

FastCGI

The module allows passing requests to a FastCGI server.

Configuration Example

```
location / {
    fastcgi_pass  localhost:9000;
    fastcgi_index index.php;

    fastcgi_param SCRIPT_FILENAME /home/www/scripts/php$fastcgi_script_name;
    fastcgi_param QUERY_STRING    $query_string;
    fastcgi_param REQUEST_METHOD  $request_method;
    fastcgi_param CONTENT_TYPE    $content_type;
    fastcgi_param CONTENT_LENGTH  $content_length;
}
```

Directives

fastcgi_bind

Syntax	`fastcgi_bind` *address* [`transparent`] | `off`;
Default	—
Context	http, server, location

Makes outgoing connections to a FastCGI server originate from the specified local IP address with an optional port. Parameter value can contain variables. The special value `off` cancels the effect of the fastcgi_bind directive inherited from the previous configuration level, which allows the system to auto-assign the local IP address and port.

The `transparent` parameter allows outgoing connections to a FastCGI server originate from a non-local IP address, for example, from a real IP address of a client:

```
fastcgi_bind $remote_addr transparent;
```

For this parameter to work, Angie worker processes usually need to run with *superuser* privileges. On Linux, this is not required: if the `transparent` parameter is specified, worker processes inherit the *CAP_NET_RAW* capability from the master process.

Important

The kernel routing table should also be configured to intercept network traffic from the FastCGI server.

fastcgi_buffer_size

Syntax	`fastcgi_buffer_size` *size*;
Default	`fastcgi_buffer_size 4k|8k;`
Context	http, server, location

Sets the size of the buffer used for reading the first part of the response received from the FastCGI server. This part usually contains a small response header. By default, the buffer size is equal to one memory page. This is either 4K or 8K, depending on a platform. It can be made smaller, however.

fastcgi_buffering

Syntax	`fastcgi_buffering on	off;`
Default	`fastcgi_buffering on;`
Context	http, server, location

Enables or disables buffering of responses from the FastCGI server.

`off`	Angie receives a response from the FastCGI server as soon as possible, saving it into the buffers set by the *fastcgi_buffer_size* and *fastcgi_buffers* directives. If the whole response does not fit into memory, a part of it can be saved to a *temporary file* on the disk. Writing to temporary files is controlled by the *fastcgi_max_temp_file_size* and *fastcgi_temp_file_write_size* directives.
`on`	the response is passed to a client synchronously, immediately as it is received. Angie will not try to read the whole response from the FastCGI server. The maximum size of the data that Angie can receive from the server at a time is set by the *fastcgi_buffer_size* directive.

Buffering can also be enabled or disabled by passing "yes" or "no" in the "X-Accel-Buffering" response header field. This capability can be disabled using the *fastcgi_ignore_headers* directive.

fastcgi_buffers

Syntax	`fastcgi_buffers` *number size*;
Default	`fastcgi_buffers 8 4k|8k;`
Context	http, server, location

Sets the number and size of the buffers used for reading a response from the FastCGI server, for a single connection.

By default, the buffer size is equal to one memory page. This is either 4K or 8K, depending on a platform.

fastcgi_busy_buffers_size

Syntax	`fastcgi_busy_buffers_size` *size*;
Default	`fastcgi_busy_buffers_size 8k	16k;`
Context	http, server, location

When *buffering* of responses from the FastCGI server is enabled, limits the total size of buffers that can be busy sending a response to the client while the response is not yet fully read. In the meantime, the rest of the buffers can be used for reading the response and, if needed, buffering part of the response to a temporary file. By default, size is limited by the size of two buffers set by the *fastcgi_buffer_size* and *fastcgi_buffers* directives.

fastcgi_cache

Syntax	`fastcgi_cache` *zone* | `off`;
Default	`fastcgi_cache off;`
Context	http, server, location

Defines a shared memory zone used for caching. The same zone can be used in several places. Parameter value can contain variables. The `off` parameter disables caching inherited from the previous configuration level.

fastcgi_cache_background_update

Syntax	`fastcgi_cache_background_update on` | `off`;
Default	`fastcgi_cache_background_update off;`
Context	http, server, location

Allows starting a background subrequest to update an expired cache item, while a stale cached response is returned to the client. Note that it is necessary to *allow* the usage of a stale cached response when it is being updated.

fastcgi_cache_bypass

Syntax	`fastcgi_cache_bypass` *string* ...;
Default	—
Context	http, server, location

Defines conditions under which the response will not be taken from a cache. If at least one value of the string parameters is not empty and is not equal to "0" then the response will not be taken from the cache:

```
fastcgi_cache_bypass $cookie_nocache $arg_nocache$arg_comment;
fastcgi_cache_bypass $http_pragma    $http_authorization;
```

Can be used along with the *fastcgi_no_cache* directive.

fastcgi_cache_key

Syntax	`fastcgi_cache_key` *string*;
Default	—
Context	http, server, location

Defines a key for caching, for example

```
fastcgi_cache_key localhost:9000$request_uri;
```

fastcgi_cache_lock

Syntax	`fastcgi_cache_lock on` | `off`;
Default	`fastcgi_cache_lock off;`
Context	http, server, location

When enabled, only one request at a time will be allowed to populate a new cache element identified according to the *fastcgi_cache_key* directive by passing a request to a FastCGI server. Other requests of the same cache element will either wait for a response to appear in the cache or the cache lock for this element to be released, up to the time set by the *fastcgi_cache_lock_timeout* directive.

fastcgi_cache_lock_age

Syntax	`fastcgi_cache_lock_age` *time*;
Default	`fastcgi_cache_lock_age 5s;`
Context	http, server, location

If the last request sent to the FastCGI server to fill a new cache entry has not completed in the specified time, another request may be sent to the FastCGI server.

fastcgi_cache_lock_timeout

Syntax	`fastcgi_cache_lock_timeout` *time*;
Default	`fastcgi_cache_lock_timeout 5s;`
Context	http, server, location

Sets a timeout for *fastcgi_cache_lock*. When the time expires, the request will be passed to the FastCGI server, however, the response will not be cached.

fastcgi_cache_max_range_offset

Syntax	`fastcgi_cache_max_range_offset` *number*;
Default	—
Context	http, server, location

Sets an offset in bytes for byte-range requests. If the range is beyond the offset, the range request will be passed to the FastCGI server and the response will not be cached.

fastcgi_cache_methods

Syntax	`fastcgi_cache_methods GET | HEAD | POST ...;`
Default	`fastcgi_cache_methods GET HEAD;`
Context	http, server, location

If the client request method is listed in this directive then the response will be cached. "GET" and "HEAD" methods are always added to the list, though it is recommended to specify them explicitly. See also the *fastcgi_no_cache* directive.

fastcgi_cache_min_uses

Syntax	`fastcgi_cache_min_uses` *number*;
Default	`fastcgi_cache_min_uses 1;`
Context	http, server, location

Sets the number of requests after which the response will be cached.

fastcgi_cache_path

Syntax	`fastcgi_cache_path` *path* [`levels=`*levels*] [`use_temp_path=on` | `off`] `keys_zone=`*name:size* [`inactive=`*time*] [`max_size=`*size*] [`min_free=`*size*] [`manager_files=`*number*] [`manager_sleep=`*time*] [`manager_threshold=`*time*] [`loader_files=`*number*] [`loader_sleep=`*time*] [`loader_threshold=`*time*];
Default	—
Context	http, server, location

Sets the path and other parameters of a cache. Cache data are stored in files. Both the key and file name in a cache are a result of applying the MD5 function to the proxied URL.

The `levels` parameter defines hierarchy levels of a cache: from 1 to 3, each level accepts values 1 or 2. For example, in the following configuration

```
fastcgi_cache_path /data/angie/cache levels=1:2 keys_zone=one:10m;
```

file names in a cache will look like this:

```
/data/angie/cache/c/29/b7f54b2df7773722d382f4809d65029c
```

A cached response is first written to a temporary file, and then the file is renamed. Temporary files and the cache can be put on different file systems. However, be aware that in this case a file is copied across two file systems instead of the cheap renaming operation. It is thus recommended that for any given location both cache and a directory holding temporary files are put on the same file system.

A directory for temporary files is set based on the `use_temp_path` parameter.

`on`	If this parameter is omitted or set to the value `on`, the directory set by the *fastcgi_temp_path* directive for the given location will be used.
`off`	temporary files will be put directly in the cache directory.

In addition, all active keys and information about data are stored in a shared memory zone, whose name and size are configured by the `keys_zone` parameter. One megabyte zone can store about 8 thousand keys.

Cached data that are not accessed during the time specified by the `inactive` parameter get removed from the cache regardless of their freshness.

By default, inactive is set to 10 minutes.

A special **cache manager** process monitors the maximum cache size, and the minimum amount of free space on the file system with cache. When the size is exceeded or there is not enough free space, it removes the least recently used data. The data is removed in iterations.

`max_size`	maximum cache size
`min_free`	minimum amount of free space on the file system with cache
`manager_files`	limits the number of items to be deleted during one iteration By default, `100`.
`manager_threshol`	limits the duration of one iteration By default, `200` milliseconds
`manager_sleep`	configures a pause between interactions By default, `50` milliseconds

A minute after Angie starts, the special **cache loader** process is activated. It loads information about previously cached data stored on file system into a cache zone. The loading is also done in iterations.

`loader_files`	limits the number of items to load during one iteration By default, `100`
`loader_threshold`	limits the duration of one iteration By default, `200` milliseconds
`loader_sleep`	configures a pause between interactions By default, `50` milliseconds

fastcgi_cache_revalidate

Syntax	`fastcgi_cache_revalidate on | off;`
Default	`fastcgi_cache_revalidate off;`
Context	http, server, location

Enables revalidation of expired cache items using conditional requests with the "If-Modified-Since" and "If-None-Match" header fields.

fastcgi_cache_use_stale

Syntax	`fastcgi_cache_use_stale error | timeout | invalid_header | updating | http_500 | http_503 | http_403 | http_429 | off ...;`
Default	`fastcgi_cache_use_stale off;`
Context	http, server, location

Determines in which cases a stale cached response can be used when an error occurs during communication with the FastCGI server. The directive's parameters match the parameters of the *fastcgi_next_upstream* directive.

`error`	permits using a stale cached response if a FastCGI server to process a request cannot be selected.
`updating`	additional parameter, permits using a stale cached response if it is currently being updated. This allows minimizing the number of accesses to FastCGI servers when updating cached data.

Using a stale cached response can also be enabled directly in the response header for a specified number of seconds after the response became stale.

- The stale-while-revalidate extension of the "Cache-Control" header field permits using a stale cached response if it is currently being updated.
- The stale-if-error extension of the "Cache-Control" header field permits using a stale cached response in case of an error.

Note

This has lower priority than using the directive parameters.

To minimize the number of accesses to FastCGI servers when populating a new cache element, the *fastcgi_cache_lock* directive can be used.

fastcgi_cache_valid

Syntax	`fastcgi_cache_valid` [*code* ...] *time*;
Default	—
Context	http, server, location

Sets caching time for different response codes. For example, the following directives

```
fastcgi_cache_valid 200 302 10m;
fastcgi_cache_valid 404      1m;
```

set 10 minutes of caching for responses with codes 200 and 302 and 1 minute for responses with code 404.

If only caching time is specified

```
fastcgi_cache_valid 5m;
```

then only 200, 301, and 302 responses are cached.

In addition, the `any` parameter can be specified to cache any responses:

```
fastcgi_cache_valid 200 302 10m;
fastcgi_cache_valid 301      1h;
fastcgi_cache_valid any      1m;
```

Note

Parameters of caching can also be set directly in the response header. This has higher priority than setting of caching time using the directive.

- The "X-Accel-Expires" header field sets caching time of a response in seconds. The zero value disables caching for a response. If the value starts with the @ prefix, it sets an absolute time in seconds since Epoch, up to which the response may be cached.
- If the header does not include the "X-Accel-Expires" field, parameters of caching may be set in the header fields "Expires" or "Cache-Control".
- If the header includes the "Set-Cookie" field, such a response will not be cached.
- If the header includes the "Vary" field with the special value "*", such a response will not be cached. If the header includes the "Vary" field with another value, such a response will be cached taking into account the corresponding request header fields.

Processing of one or more of these response header fields can be disabled using the *fastcgi_ignore_headers* directive.

fastcgi_catch_stderr

Syntax	`fastcgi_catch_stderr` *string*;
Default	—
Context	http, server, location

Sets a string to search for in the error stream of a response received from a FastCGI server. If the string is found then it is considered that the FastCGI server has returned an *invalid response*. This allows handling application errors in Angie, for example:

```
location /php/ {
    fastcgi_pass backend:9000;
    ...
    fastcgi_catch_stderr "PHP Fatal error";
    fastcgi_next_upstream error timeout invalid_header;
}
```

fastcgi_connect_timeout

Syntax	`fastcgi_connect_timeout` *time*;
Default	`fastcgi_connect_timeout 60s;`
Context	http, server, location

Defines a timeout for establishing a connection with a FastCGI server. It should be noted that this timeout cannot usually exceed 75 seconds.

fastcgi_connection_drop

Syntax	`fastcgi_connection_drop` *time* | `on` | `off`;
Default	`fastcgi_connection_drop off;`
Context	http, server, location

Enables termination of all connections to the proxied server after it has been removed from the group or marked as permanently unavailable by a *reresolve* process or the *API command* `DELETE`.

A connection is terminated when the next read or write event is processed for either the client or the proxied server.

Setting *time* enables a connection termination *timeout*; with `on` set, connections are dropped immediately.

fastcgi_force_ranges

Syntax	`fastcgi_force_ranges on` | `off`;
Default	`fastcgi_force_ranges off;`
Context	http, server, location

Enables byte-range support for both cached and uncached responses from the FastCGI server regardless of the "Accept-Ranges" field in these responses.

fastcgi_hide_header

Syntax	`fastcgi_hide_header` *field*;
Default	—
Context	http, server, location

By default, Angie does not pass the header fields "Status" and "X-Accel-..." from the response of a FastCGI server to a client. The *fastcgi_hide_header* directive sets additional fields that will not be passed. If, on the contrary, the passing of fields needs to be permitted, the *fastcgi_pass_header* directive can be used.

fastcgi_ignore_client_abort

Syntax	`fastcgi_ignore_client_abort on | off;`
Default	`fastcgi_ignore_client_abort off;`
Context	http, server, location

Determines whether the connection with a FastCGI server should be closed when a client closes the connection without waiting for a response.

fastcgi_ignore_headers

Syntax	`fastcgi_ignore_headers` *field*;
Default	—
Context	http, server, location

Disables processing of certain response header fields from the FastCGI server. The following fields can be ignored: "X-Accel-Redirect", "X-Accel-Expires", "X-Accel-Limit-Rate", "X-Accel-Buffering", "X-Accel-Charset", "Expires", "Cache-Control", "Set-Cookie", and "Vary".

If not disabled, processing of these header fields has the following effect:

- "X-Accel-Expires", "Expires", "Cache-Control", "Set-Cookie", and "Vary" set the *parameters* of response caching;
- "X-Accel-Redirect" performs an *internal* redirect to the specified URI;
- "X-Accel-Limit-Rate" sets the *rate limit* for transmission of a response to a client;
- "X-Accel-Buffering" enables or disables *buffering* of a response;
- "X-Accel-Charset" sets the desired *charset* of a response.

fastcgi_index

Syntax	`fastcgi_index` *name*;
Default	—
Context	http, server, location

Sets a file name that will be appended after a URI that ends with a slash, in the value of the *$fastcgi_script_name* variable. For example, with these settings

```
fastcgi_index index.php;
fastcgi_param SCRIPT_FILENAME /home/www/scripts/php$fastcgi_script_name;
```

and the `/page.php` request, the SCRIPT_FILENAME parameter will be equal to `/home/www/scripts/php/page.php`,

and with the `/` request it will be equal to `/home/www/scripts/php/index.php`.

fastcgi_intercept_errors

Syntax	`fastcgi_intercept_errors on | off;`
Default	`fastcgi_intercept_errors off;`
Context	http, server, location

Determines whether FastCGI server responses with codes greater than or equal to 300 should be passed to a client or be intercepted and redirected to Angie for processing with the *error_page* directive.

fastcgi_keep_conn

Syntax	`fastcgi_keep_conn on | off;`
Default	`fastcgi_keep_conn off;`
Context	http, server, location

By default, a FastCGI server will close a connection right after sending the response. However, when this directive is set to the value `on`, Angie will instruct a FastCGI server to keep connections open. This is necessary, in particular, for *keepalive* connections to FastCGI servers to function.

fastcgi_limit_rate

Syntax	`fastcgi_limit_rate` *rate*;
Default	`fastcgi_limit_rate 0;`
Context	http, server, location

Limits the speed of reading the response from the FastCGI server. The *rate* is specified in bytes per second and can contain variables.

`0`	disables rate limiting

Note

The limit is set per a request, and so if Angie simultaneously opens two connections to the FastCGI server, the overall rate will be twice as much as the specified limit. The limitation works only if *buffering* of responses from the FastCGI server is enabled.

fastcgi_max_temp_file_size

Syntax	`fastcgi_max_temp_file_size` *size*;
Default	`fastcgi_max_temp_file_size 1024m;`
Context	http, server, location

When *buffering* of responses from the FastCGI server is enabled, and the whole response does not fit into the buffers set by the *fastcgi_buffer_size* and *fastcgi_buffers* directives, a part of the response can be saved to a temporary file. This directive sets the maximum size of the temporary file. The size of data written to the temporary file at a time is set by the *fastcgi_temp_file_write_size* directive.

`0`	disables buffering of responses to temporary files

Note

This restriction does not apply to responses that will be *cached* or stored on *disk*.

fastcgi_next_upstream

Syntax	`fastcgi_next_upstream error | timeout | invalid_header | http_500 | http_503 | http_403 | http_404 | http_429 | non_idempotent | off ...;`
Default	`fastcgi_next_upstream error timeout;`
Context	http, server, location

Specifies in which cases a request should be passed to the next server:

`error`	an error occurred while establishing a connection with the server, passing a request to it, or reading the response header;
`timeout`	a timeout has occurred while establishing a connection with the server, passing a request to it, or reading the response header;
`invalid_header`	a server returned an empty or invalid response;
`http_500`	a server returned a response with the code 500;
`http_503`	a server returned a response with the code 503;
`http_403`	a server returned a response with the code 403;
`http_404`	a server returned a response with the code 404;
`http_429`	a server returned a response with the code 429;
`non_idempotent`	normally, requests with a non-idempotent method (`POST`, `LOCK`, `PATCH`) are not passed to the next server if a request has been sent to an upstream server; enabling this option explicitly allows retrying such requests;
`off`	disables passing a request to the next server.

Note

One should bear in mind that passing a request to the next server is only possible if nothing has been sent to a client yet. That is, if an error or timeout occurs in the middle of the transferring of a response, fixing this is impossible.

The directive also defines what is considered an *unsuccessful attempt* of communication with a server.

`error`
`timeout`
`invalid_header`
	always considered unsuccessful attempts, even if they are not specified in the directive

`http_500`
`http_503`
`http_429`
	considered unsuccessful attempts only if they are specified in the directive

`http_403` `http_404`	never considered unsuccessful attempts

Passing a request to the next server can be limited by the *number of tries* and by *time*.

fastcgi_next_upstream_timeout

Syntax	`fastcgi_next_upstream_timeout` *time*;
Default	`fastcgi_next_upstream_timeout 0;`
Context	http, server, location

Limits the time during which a request can be passed to the *next server*.

`0`	turns off this limitation

fastcgi_next_upstream_tries

Syntax	`fastcgi_next_upstream_tries` *number*;
Default	`fastcgi_next_upstream_tries 0;`
Context	http, server, location

Limits the number of possible tries for passing a request to the *next server*.

`0`	turns off this limitation

fastcgi_no_cache

Syntax	`fastcgi_no_cache` *string* ...;
Default	—
Context	http, server, location

Defines conditions under which the response will not be saved to a cache. If at least one value of the string parameters is not empty and is not equal to "0" then the response will not be saved:

```
fastcgi_no_cache $cookie_nocache $arg_nocache$arg_comment;
fastcgi_no_cache $http_pragma    $http_authorization;
```

Can be used along with the *fastcgi_cache_bypass* directive.

fastcgi_param

Syntax	`fastcgi_param` *parameter value* [`if_not_empty`];
Default	—
Context	http, server, location

Sets a parameter that should be passed to the FastCGI server. The value can contain text, variables, and their combination. These directives are inherited from the previous configuration level if and only if there are no *fastcgi_param* directives defined on the current level.

The following example shows the minimum required settings for PHP:

```
fastcgi_param SCRIPT_FILENAME /home/www/scripts/php$fastcgi_script_name;
fastcgi_param QUERY_STRING    $query_string;
```

The SCRIPT_FILENAME parameter is used in PHP for determining the script name, and the QUERY_STRING parameter is used to pass request parameters.

For scripts that process POST requests, the following three parameters are also required:

```
fastcgi_param REQUEST_METHOD  $request_method;
fastcgi_param CONTENT_TYPE    $content_type;
fastcgi_param CONTENT_LENGTH  $content_length;
```

If PHP was built with the *--enable-force-cgi-redirect* configuration parameter, the REDIRECT_STATUS parameter should also be passed with the value "200":

```
fastcgi_param REDIRECT_STATUS 200;
```

If the directive is specified with `if_not_empty` then such a parameter will be passed to the server only if its value is not empty:

```
fastcgi_param HTTPS            $https if_not_empty;
```

fastcgi_pass

Syntax	`fastcgi_pass` *address*;
Default	—
Context	location, if in location

Sets the address of a FastCGI server. The address can be specified as a domain name or IP address, and a port:

```
fastcgi_pass localhost:9000;
```

or as a UNIX domain socket path:

```
fastcgi_pass unix:/tmp/fastcgi.socket;
```

If a domain name resolves to several addresses, all of them will be used in a round-robin fashion. In addition, an address can be specified as a *server group*. If a group is used, you cannot specify the port with it; instead, specify the port for each server within the group individually.

Parameter value can contain variables. In this case, if an address is specified as a domain name, the name is searched among the described *server groups*, and, if not found, is determined using a *resolver*.

fastcgi_pass_header

Syntax	`fastcgi_pass_header` *field*;
Default	—
Context	http, server, location

Permits passing *otherwise disabled* header fields from a FastCGI server to a client.

fastcgi_pass_request_body

Syntax	`fastcgi_pass_request_body on | off`;
Default	—
Context	http, server, location

Indicates whether the original request body is passed to the FastCGI server. See also the *fastcgi_pass_request_headers* directive.

fastcgi_pass_request_headers

Syntax	`fastcgi_pass_request_headers on | off`;
Default	—
Context	http, server, location

Indicates whether the header fields of the original request are passed to the FastCGI server. See also the *fastcgi_pass_request_body* directive.

fastcgi_read_timeout

Syntax	`fastcgi_read_timeout` *time*;
Default	`fastcgi_read_timeout 60s;`
Context	http, server, location

Defines a timeout for reading a response from the FastCGI server. The timeout is set only between two successive read operations, not for the transmission of the whole response. If the FastCGI server does not transmit anything within this time, the connection is closed.

fastcgi_request_buffering

Syntax	`fastcgi_request_buffering on` | `off`;
Default	`fastcgi_request_buffering on;`
Context	http, server, location

Enables or disables buffering of a client request body.

`on`	the entire request body is *read* from the client before sending the request to a FastCGI server.
`off`	the request body is sent to the FastCGI server immediately as it is received. In this case, the request cannot be passed to the *next server*, if Angie already started sending the request body.

fastcgi_send_lowat

Syntax	`fastcgi_send_lowat` *size*;
Default	`fastcgi_send_lowat 0;`
Context	http, server, location

If the directive is set to a non-zero value, Angie will try to minimize the number of send operations on outgoing connections to a FastCGI server by using either *NOTE_LOWAT* flag of the *kqueue* method, or the *SO_SNDLOWAT* socket option, with the specified size.

> **Note**
>
> This directive is ignored on Linux, Solaris, and Windows.

fastcgi_send_timeout

Syntax	`fastcgi_send_timeout` *time*;
Default	`fastcgi_send_timeout 60s;`
Context	http, server, location

Sets a timeout for transmitting a request to the FastCGI server. The timeout is set only between two successive write operations, not for the transmission of the whole request. If the FastCGI server does not receive anything within this time, the connection is closed.

fastcgi_socket_keepalive

Syntax	`fastcgi_socket_keepalive on	off;`
Default	`fastcgi_socket_keepalive off;`
Context	http, server, location

Configures the "TCP keepalive" behavior for outgoing connections to a FastCGI server.

`""`	By default, the operating system's settings are in effect for the socket.
`on`	the *SO_KEEPALIVE* socket option is turned on for the socket.

fastcgi_split_path_info

Syntax	`fastcgi_split_path_info` *regex*;
Default	—
Context	location

Defines a regular expression that captures a value for the *$fastcgi_path_info* variable. The regular expression should have two captures: the first becomes a value of the *$fastcgi_script_name* variable, the second becomes a value of the *$fastcgi_path_info* variable. For example, with these settings

```
location ~ ^(.+\.php)(.*)$ {
    fastcgi_split_path_info       ^(.+\.php)(.*)$;
    fastcgi_param SCRIPT_FILENAME /path/to/php$fastcgi_script_name;
    fastcgi_param PATH_INFO       $fastcgi_path_info;
```

and the `/show.php/article/0001` request, the `SCRIPT_FILENAME` parameter will be equal to `/path/to/php/show.php`, and the `PATH_INFO` parameter will be equal to `/article/0001`.

fastcgi_store

Syntax	`fastcgi_store on	off |` *string*;
Default	`fastcgi_store off;`
Context	http, server, location

Enables saving of files to a disk.

`on`	saves files with paths corresponding to the directives *alias* or *root*.
`off`	disables saving of files

In addition, the file name can be set explicitly using the string with variables:

```
fastcgi_store /data/www$original_uri;
```

The modification time of files is set according to the received "Last-Modified" response header field. The response is first written to a temporary file, and then the file is renamed. Temporary files and the persistent store can be put on different file systems. However, be aware that in this case a file is copied across two file systems instead of the cheap renaming operation. It is thus recommended that for any given location both saved files and a directory holding temporary files, set by the *fastcgi_temp_path* directive, are put on the same file system.

This directive can be used to create local copies of static unchangeable files, e.g.:

```
location /images/ {
    root                /data/www;
    error_page          404 = /fetch$uri;
}

location /fetch/ {
    internal;

    fastcgi_pass        backend:9000;
    ...

    fastcgi_store       on;
    fastcgi_store_access user:rw group:rw all:r;
    fastcgi_temp_path   /data/temp;

    alias               /data/www/;
}
```

fastcgi_store_access

Syntax	`fastcgi_store_access` users:permissions ...;
Default	`fastcgi_store_access user:rw;`
Context	http, server, location

Sets access permissions for newly created files and directories, e.g.:

```
fastcgi_store_access user:rw group:rw all:r;
```

If any group or all access permissions are specified then user permissions may be omitted:

```
fastcgi_store_access group:rw all:r;
```

fastcgi_temp_file_write_size

Syntax	`fastcgi_temp_file_write_size` *size*;
Default	`fastcgi_temp_file_write_size 8k|16k;`
Context	http, server, location

Limits the size of data written to a temporary file at a time, when buffering of responses from the FastCGI server to temporary files is enabled. By default, size is limited by two buffers set by the *fastcgi_buffer_size* and *fastcgi_buffers* directives. The maximum size of a temporary file is set by the *fastcgi_max_temp_file_size* directive.

fastcgi_temp_path

Syntax	`fastcgi_temp_path` *path* [*level1* [*level2* [*level3*]]]`;
Default	`fastcgi_temp_path fastcgi_temp;` (the path depends on the `--http-fastcgi-temp-path` build option)
Context	http, server, location

Defines a directory for storing temporary files with data received from FastCGI servers. Up to three-level subdirectory hierarchy can be used underneath the specified directory. For example, in the following configuration

```
fastcgi_temp_path /spool/angie/fastcgi_temp 1 2;
```

a temporary file might look like this:

```
/spool/angie/fastcgi_temp/7/45/00000123457
```

See also the *use_temp_path* parameter of the *fastcgi_cache_path* directive.

Parameters Passed to a FastCGI Server

HTTP request header fields are passed to a FastCGI server as parameters. In applications and scripts running as FastCGI servers, these parameters are usually made available as environment variables. For example, the "User-Agent" header field is passed as the HTTP_USER_AGENT parameter. In addition to HTTP request header fields, it is possible to pass arbitrary parameters using the *fastcgi_param* directive.

Built-in Variables

The *http_fastcgi* module supports built-in variables that can be used to set parameters using the *fastcgi_param* directive:

`$fastcgi_script_name`

request URI or, if a URI ends with a slash, request URI with an index file name configured by the *fastcgi_index* directive appended to it. This variable can be used to set the SCRIPT_FILENAME and PATH_TRANSLATED parameters that determine the script name in PHP. For example, for the `/info/` request with the following directives

```
fastcgi_index index.php;
fastcgi_param SCRIPT_FILENAME /home/www/scripts/php$fastcgi_script_name;
```

the SCRIPT_FILENAME parameter will be equal to `/home/www/scripts/php/info/index.php`.

When using the *fastcgi_split_path_info* directive, the *$fastcgi_script_name* variable equals the value of the first capture set by the directive.

`$fastcgi_path_info`

the value of the second capture set by the *fastcgi_split_path_info* directive. This variable can be used to set the PATH_INFO parameter.

FLV

The module provides pseudo-streaming server-side support for Flash Video (FLV) files.

It handles requests with the start argument in the request URI's query string specially, by sending back the contents of a file starting from the requested byte offset and with the prepended FLV header.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-http_flv_module` build option.

In packages and images from our repos, the module is included in the build.

Configuration Example

```
location ~ \.flv$ {
    flv;
}
```

Directives

flv

Syntax	`flv;`
Default	—
Context	location

Turns on module processing in a surrounding location.

Geo

The module creates variables with values depending on the client IP address.

Configuration Example

```
geo $geo {
    default         0;

    127.0.0.1       2;
    192.168.1.0/24 1;
    10.1.0.0/16     1;

    ::1             2;
    2001:0db8::/32 1;
}
```

Directives

geo

Syntax	`geo` [*$address*] *$variable* { ... }
Default	—
Context	http

Describes the dependency of values of the specified variable on the client IP address. By default, the address is taken from the *$remote_addr* variable, but it can also be taken from another variable, for example:

```
geo $arg_remote_addr $geo {
    ...;
}
```

Note

Since variables are evaluated only when used, the mere existence of even a large number of declared `geo` variables does not cause any extra costs for request processing.

If the value of a variable does not represent a valid IP address then the "255.255.255.255" address is used.

Addresses are specified either as prefixes in CIDR notation (including individual addresses) or as ranges.

The following special parameters are also supported:

`delete`	deletes the specified network
`default`	the value set to the variable if the client address does not match any of the specified addresses. When addresses are specified in CIDR notation, `0.0.0.0/0` and `::/0` can be used instead of `default`. When `default` is not specified, the default value will be an empty string
`include`	includes a file with addresses and values. There can be several inclusions.
`proxy`	defines trusted addresses. When a request comes from a trusted address, an address from the `X-Forwarded-For` request header field will be used instead. In contrast to the regular addresses, trusted addresses are checked sequentially.
`proxy_recursive`	enables recursive address search. If recursive search is disabled then instead of the original client address that matches one of the trusted addresses, the last address sent in `X-Forwarded-For` will be used. If recursive search is enabled then instead of the original client address that matches one of the trusted addresses, the last non-trusted address sent in `X-Forwarded-For` will be used.
`ranges`	indicates that addresses are specified as ranges. This parameter should be the first. To speed up loading of a geo base, addresses should be put in ascending order.

Example:

```
geo $country {
    default         ZZ;
    include         conf/geo.conf;
    delete          127.0.0.0/16;
    proxy           192.168.100.0/24;
    proxy           2001:0db8::/32;

    127.0.0.0/24    US;
    127.0.0.1/32    RU;
    10.1.0.0/16     RU;
    192.168.1.0/24 UK;
}
```

The `conf/geo.conf` file could contain the following lines:

```
10.2.0.0/16     RU;
192.168.2.0/24 RU;
```

The value of the most specific match is used. For example, for the `127.0.0.1` address, the value `RU` will be chosen, not `US`.

Sample range description:

```
geo $country {
    ranges;
    default                    ZZ;
    127.0.0.0-127.0.0.0        US;
    127.0.0.1-127.0.0.1        RU;
    127.0.0.2-127.0.0.255      US;
    10.1.0.0-10.1.255.255      RU;
    192.168.1.0-192.168.1.255 UK;
}
```

GeoIP

Creates variables with values depending on the client IP address, using the precompiled MaxMind databases or their counterparts.

When using the databases with IPv6 support, IPv4 addresses are looked up as IPv4-mapped IPv6 addresses.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-http_geoip_module` build option.

Important

This module requires the MaxMind GeoIP database or a counterpart such as MaxMind GeoLite2.

Configuration Example

```
http {
    geoip_country         GeoIP.dat;
    geoip_city            GeoLiteCity.dat;
    geoip_proxy           192.168.100.0/24;
    geoip_proxy           2001:0db8::/32;
    geoip_proxy_recursive on;
    ...
```

Directives

geoip_country

Syntax	`geoip_country` *file*;
Default	—
Context	http

Specifies a database used to determine the country depending on the client IP address. The following variables are available when using this database:

`$geoip_country_c`	two-letter country code, for example, "RU", "US".
`$geoip_country_c`	three-letter country code, for example, "RUS", "USA".
`$geoip_country_n`	country name, for example, "Russian Federation", "United States".

geoip_city

Syntax	`geoip_city` *file*;
Default	—
Context	http

Specifies a database used to determine the country, region, and city depending on the client IP address. The following variables are available when using this database:

`$geoip_city_cont`	two-letter continent code, for example, "EU", "NA".
`$geoip_city_coun`	two-letter country code, for example, "RU", "US".
`$geoip_city_coun`	three-letter country code, for example, "RUS", "USA".
`$geoip_city_coun`	country name, for example, "Russian Federation", "United States".
`$geoip_dma_code`	DMA region code in US (also known as "metro code"), according to the geotargeting in Google AdWords API.
`$geoip_latitude`	latitude.
`$geoip_longitude`	longitude.
`$geoip_region`	two-symbol country region code (region, territory, state, province, federal land and the like), for example, "48", "DC".
`$geoip_region_na`	country region name (region, territory, state, province, federal land and the like), for example, "Moscow City", "District of Columbia".
`$geoip_city`	city name, for example, "Moscow", "Washington".
`$geoip_postal_co`	postal code.

geoip_org

Syntax	`geoip_org` *file*;
Default	—
Context	http

Specifies a database used to determine the organization depending on the client IP address. The following variable is available when using this database:

`$geoip_org`	organization name, for example, "The University of Melbourne".

geoip_proxy

Syntax	`geoip_proxy` *file*;
Default	—
Context	http

Defines trusted addresses. When a request comes from a trusted address, an address from the `X-Forwarded-For` request header field will be used instead.

geoip_proxy_recursive

Syntax	`geoip_proxy_recursive on` | `off`;
Default	`geoip_proxy_recursive off;`
Context	http

If recursive search is disabled then instead of the original client address that matches one of the trusted addresses, the last address sent in `X-Forwarded-For` will be used. If recursive search is enabled then instead of the original client address that matches one of the trusted addresses, the last non-trusted address sent in `X-Forwarded-For` will be used.

gRPC

Allows passing requests to a gRPC server. The module requires the *HTTP/2*.

Configuration Example

```
server {
    listen 9000;

    http2 on;

    location / {
        grpc_pass 127.0.0.1:9000;
    }
}
```

Directives

grpc_bind

Syntax	`grpc_bind` *address* [`transparent`] | `off`;
Default	—
Context	http, server, location

Makes outgoing connections to a gRPC server originate from the specified local IP address with an optional port. Parameter value can contain variables. The special value `off` cancels the effect of the *grpc_bind* directive inherited from the previous configuration level, which allows the system to auto-assign the local IP address and port.

The `transparent` parameter allows outgoing connections to a gRPC server originate from a non-local IP address, for example, from a real IP address of a client:

```
grpc_bind $remote_addr transparent;
```

In order for this parameter to work, it is usually necessary to run Angie worker processes with the *superuser* privileges. On Linux it is not required as if the `transparent` parameter is specified, worker processes inherit the *CAP_NET_RAW* capability from the master process.

Important

It is necessary to configure kernel routing table to intercept network traffic from the gRPC server.

grpc_buffer_size

Syntax	`grpc_buffer_size` *size*;
Default	`grpc_buffer_size 4k|8k;`
Context	http, server, location

Sets the size of the buffer used for reading the first part of the response received from the gRPC server. The response is passed to the client synchronously, as soon as it is received.

grpc_connect_timeout

Syntax	`grpc_connect_timeout` *time*;
Default	`grpc_connect_timeout 60s;`
Context	http, server, location

Defines a timeout for establishing a connection with a gRPC server. It should be noted that this timeout cannot usually exceed 75 seconds.

grpc_connection_drop

Syntax	`grpc_connection_drop` *time* | `on` | `off`;
Default	`grpc_connection_drop off;`
Context	http, server, location

Enables termination of all connections to the proxied server after it has been removed from the group or marked as permanently unavailable by a *reresolve* process or the *API command* `DELETE`.

A connection is terminated when the next read or write event is processed for either the client or the proxied server.

Setting *time* enables a connection termination *timeout*; with `on` set, connections are dropped immediately.

grpc_hide_header

Syntax	`grpc_hide_header` *field*;
Default	—
Context	http, server, location

By default, Angie does not pass the header fields "Date", "Server", and "X-Accel-..." from the response of a gRPC server to a client. The *grpc_hide_header* directive sets additional fields that will not be passed. If, on the contrary, the passing of fields needs to be permitted, the *grpc_pass_header* directive can be used.

grpc_ignore_headers

Syntax	`grpc_ignore_headers` *field* ...;
Default	—
Context	http, server, location

Disables processing of certain response header fields from the gRPC server. The following fields can be ignored: "X-Accel-Redirect" and "X-Accel-Charset".

If not disabled, processing of these header fields has the following effect:

- "X-Accel-Redirect" performs an *internal* redirect to the specified URI;
- "X-Accel-Charset" sets the desired *charset* of a response.

grpc_intercept_errors

Syntax	`grpc_intercept_errors on` | `off`;
Default	`grpc_intercept_errors off;`
Context	http, server, location

Determines whether gRPC responses with codes greater than or equal to 300 should be passed to a client or be intercepted and redirected to Angie for processing with the *error_page* directive.

grpc_next_upstream

Syntax	`grpc_next_upstream error | timeout | invalid_header | http_500 | http_502 | http_503 | http_504 | http_403 | http_404 | http_429 | non_idempotent | off ...;`
Default	`grpc_next_upstream error timeout;`
Context	http, server, location

Specifies in which cases a request should be passed to the next server in the *upstream pool*:

`error`	an error occurred while establishing a connection with the server, passing a request to it, or reading the response header;
`timeout`	a timeout has occurred while establishing a connection with the server, passing a request to it, or reading the response header;
`invalid_header`	a server returned an empty or invalid response;
`http_500`	a server returned a response with the code 500;
`http_502`	a server returned a response with the code 502;
`http_503`	a server returned a response with the code 503;
`http_504`	a server returned a response with the code 504;
`http_403`	a server returned a response with the code 403;
`http_404`	a server returned a response with the code 404;
`http_429`	a server returned a response with the code 429;
`non_idempotent`	normally, requests with a non-idempotent method (`POST`, `LOCK`, `PATCH`) are not passed to the next server if a request has been sent to an upstream server; enabling this option explicitly allows retrying such requests;
`off`	disables passing a request to the next server.

Note

One should bear in mind that passing a request to the next server is only possible if nothing has been sent to a client yet. That is, if an error or timeout occurs in the middle of the transferring of a response, fixing this is impossible.

The directive also defines what is considered an *unsuccessful attempt* of communication with a server.

`error`, `timeout`, `invalid_header`	always considered unsuccessful attempts, even if they are not specified in the directive
`http_500`, `http_502`, `http_503`, `http_504`, `http_429`	considered unsuccessful attempts only if they are specified in the directive
`http_403`, `http_404`	never considered unsuccessful attempts

Passing a request to the next server can be limited by the *number of tries* and by *time*.

grpc_next_upstream_timeout

Syntax	`grpc_next_upstream_timeout` *time*;
Default	`grpc_next_upstream_timeout 0;`
Context	http, server, location

Limits the time during which a request can be passed to the *next* server.

`0`	turns off this limitation

grpc_next_upstream_tries

Syntax	`grpc_next_upstream_tries` *number*;
Default	`grpc_next_upstream_tries 0;`
Context	http, server, location

Limits the number of possible tries for passing a request to the *next* server.

`0`	turns off this limitation

grpc_pass

Syntax	`grpc_pass` *address*;
Default	—
Context	location, if in location

Sets gRPC server address. The address can be specified as a domain name or IP address, and a port:

```
grpc_pass localhost:9000;
```

or as a UNIX domain socket path:

```
grpc_pass unix:/tmp/grpc.socket;
```

Alternatively, the `grpc://` scheme can be used:

```
grpc_pass grpc://127.0.0.1:9000;
```

To use gRPC over SSL, the `grpcs://` scheme should be used:

```
grpc_pass grpcs://127.0.0.1:443;
```

If a domain name resolves to several addresses, all of them will be used in a round-robin fashion. In addition, an address can be specified as a *server group*.

Parameter value can contain variables. In this case, if an address is specified as a domain name, the name is searched among the described server groups, and, if not found, is determined using a *resolver*.

grpc_pass_header

Syntax	`grpc_pass_header` *field*;
Default	—
Context	http, server, location

Permits passing *otherwise disabled* header fields from a gRPC server to a client.

grpc_read_timeout

Syntax	`grpc_read_timeout` *time*;
Default	`grpc_read_timeout 60s;`
Context	http, server, location

Defines a timeout for reading a response from the gRPC server. The timeout is set only between two successive read operations, not for the transmission of the whole response. If the gRPC server does not transmit anything within this time, the connection is closed.

grpc_send_timeout

Syntax	`grpc_send_timeout` *time*;
Default	`grpc_send_timeout 60s;`
Context	http, server, location

Sets a timeout for transmitting a request to the gRPC server. The timeout is set only between two successive write operations, not for the transmission of the whole request. If the gRPC server does not receive anything within this time, the connection is closed.

grpc_set_header

Syntax	`grpc_set_header` *field value*;
Default	`grpc_set_header Content-Length $content_length;`
Context	http, server, location

Allows redefining or appending fields to the request header *passed* to the gRPC server. The value can contain text, variables, and their combinations. These directives are inherited from the previous configuration level if and only if there are no grpc_set_header directives defined on the current level.

If the value of a header field is an empty string then this field will not be passed to a gRPC server:

```
grpc_set_header Accept-Encoding "";
```

grpc_socket_keepalive

Syntax	`grpc_socket_keepalive on | off;`
Default	`grpc_socket_keepalive off;`
Context	http, server, location

Configures the "TCP keepalive" behavior for outgoing connections to a gRPC server.

`""`	By default, the operating system's settings are in effect for the socket.
`on`	The *SO_KEEPALIVE* socket option is turned on for the socket.

grpc_ssl_certificate

Syntax	`grpc_ssl_certificate` *file*;
Default	—
Context	http, server, location

Specifies a file with the certificate in the PEM format used for authentication to a gRPC SSL server. Variables can be used in the file name.

grpc_ssl_certificate_cache

Syntax	`grpc_ssl_certificate_cache off`; `grpc_ssl_certificate_cache max=`*N* [`inactive=`*time*] [`valid=`*time*];
Default	`grpc_ssl_certificate_cache off;`
Context	http, server, location

Defines a cache that stores *SSL certificates* and *secret keys* specified using variables.

The directive supports the following parameters:

- `max` — sets the maximum number of elements in the cache. When the cache overflows, the least recently used (LRU) elements are removed.
- `inactive` — defines the time after which an element is removed if it has not been accessed. The default is 10 seconds.
- `valid` — defines the time during which a cached element is considered valid and can be reused. The default is 60 seconds. After this period, certificates are reloaded or revalidated.
- `off` — disables the cache.

Example:

```
grpc_ssl_certificate       $grpc_ssl_server_name.crt;
grpc_ssl_certificate_key   $grpc_ssl_server_name.key;
grpc_ssl_certificate_cache max=1000 inactive=20s valid=1m;
```

grpc_ssl_certificate_key

Syntax	`grpc_ssl_certificate_key` *file*;
Default	—
Context	http, server, location

Specifies a file with the secret key in the PEM format used for authentication to a gRPC SSL server.

The value "engine:*name*:id" can be specified instead of the file, which loads a secret key with a specified id from the OpenSSL engine name. Variables can be used in the file name.

grpc_ssl_ciphers

Syntax	`grpc_ssl_ciphers` *ciphers*;
Default	`grpc_ssl_ciphers DEFAULT;`
Context	http, server, location

Specifies the enabled ciphers for requests to a gRPC SSL server. The ciphers are specified in the format understood by the OpenSSL library.

The list of ciphers depends on the version of OpenSSL installed. The full list can be viewed using the `openssl ciphers` command.

Attention

The `grpc_ssl_ciphers` directive does *not* configure ciphers for TLS 1.3 when using OpenSSL. To tune TLS 1.3 ciphers with OpenSSL, use the *grpc_ssl_conf_command* directive, which was added to support advanced SSL configuration.

- In LibreSSL, TLS 1.3 ciphers *can* be configured using `grpc_ssl_ciphers`.
- In BoringSSL, TLS 1.3 ciphers cannot be configured at all.

grpc_ssl_conf_command

Syntax	`grpc_ssl_conf_command` *name value*;
Default	—
Context	http, server, location

Sets arbitrary OpenSSL configuration commands when establishing a connection with the gRPC SSL server.

Important

The directive is supported when using OpenSSL 1.0.2 or higher. To configure TLS 1.3 ciphers with OpenSSL, use the `ciphersuites` command.

Several *grpc_ssl_conf_command* directives can be specified on the same level. These directives are inherited from the previous configuration level if and only if there are no *grpc_ssl_conf_command* directives defined on the current level.

Caution

Note that configuring OpenSSL directly might result in unexpected behavior.

grpc_ssl_crl

Syntax	`grpc_ssl_crl` *file*;
Default	—
Context	http, server, location

Specifies a file with revoked certificates (CRL) in the PEM format used to *verify* the certificate of the gRPC SSL server.

grpc_ssl_name

Syntax	`grpc_ssl_name` *name*;
Default	`grpc_ssl_name` `host from grpc_pass;`
Context	http, server, location

Allows overriding the server name used to *verify* the certificate of the gRPC SSL server and to be *passed through SNI* when establishing a connection with the gRPC SSL server.

By default, the host part of the *grpc_pass* URL is used.

grpc_ssl_password_file

Syntax	`grpc_ssl_password_file` *file*;
Default	—
Context	http, server, location

Specifies a file with passphrases for *secret keys* where each passphrase is specified on a separate line. Passphrases are tried in turn when loading the key.

grpc_ssl_protocols

Syntax	`grpc_ssl_protocols` [SSLv2] [SSLv3] [TLSv1] [TLSv1.1] [TLSv1.2] [TLSv1.3];
Default	`grpc_ssl_protocols TLSv1.2 TLSv1.3;`
Context	http, server, location

Changed in version 1.2.0: `TLSv1.3` parameter added to default set.

Enables the specified protocols for requests to a gRPC HTTPS server.

grpc_ssl_server_name

Syntax	`grpc_ssl_server_name on` | `off`;
Default	`grpc_ssl_server_name off;`
Context	http, server, location

Enables or disables passing the server name set by the *grpc_ssl_name* directive via the Server Name Indication TLS extension (SNI, RFC 6066) while establishing a connection with the gRPC SSL server.

grpc_ssl_session_reuse

Syntax	`grpc_ssl_session_reuse on` | `off`;
Default	`grpc_ssl_session_reuse on;`
Context	http, server, location

Determines whether SSL sessions can be reused when working with the gRPC server. If the errors "*SSL3_GET_FINISHED:digest check failed*" appear in the logs, try disabling session reuse.

grpc_ssl_trusted_certificate

Syntax	`grpc_ssl_trusted_certificate` *file*;
Default	—
Context	http, server, location

Specifies a file with trusted CA certificates in the PEM format used to *verify* the certificate of the gRPC SSL server.

grpc_ssl_verify

Syntax	`grpc_ssl_verify on | off;`
Default	`grpc_ssl_verify off;`
Context	http, server, location

Enables or disables verification of the gRPC SSL server certificate.

grpc_ssl_verify_depth

Syntax	`grpc_ssl_verify_depth` *number*;
Default	`grpc_ssl_verify_depth 1;`
Context	http, server, location

Sets the verification depth in the gRPC SSL server certificates chain.

GunZIP

The module is a filter that decompresses responses with "Content-Encoding: gzip" for clients that do not support "gzip" encoding method. The module will be useful when it is desirable to store data compressed to save space and reduce I/O costs.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-http_gunzip_module` build option.

In packages and images from our repos, the module is included in the build.

Configuration Example

```
location /storage/ {
    gunzip on;
#    ...
}
```

Directives

gunzip

Syntax	`gunzip on | off;`
Default	`gunzip off;`
Context	http, server, location

Enables or disables decompression of gzipped responses for clients that lack gzip support. If enabled, the following directives are also taken into account when determining if clients support gzip: *gzip_http_version*, *gzip_proxied* and *gzip_disable*. See also the *gzip_vary* directive.

gunzip_buffers

Syntax	`gunzip_buffers` *number size*;
Default	`gunzip_buffers 32 4k | 16 8k;`
Context	http, server, location

Sets the number and size of buffers used to decompress a response. By default, the buffer size is equal to one memory page. This is either 4K or 8K, depending on a platform.

GZip

The module is a filter that compresses responses using the "gzip" method. This often helps to reduce the size of transmitted data by half or even more.

> **Caution**
>
> When using the SSL/TLS protocol, compressed responses may be subject to BREACH attacks.

Configuration Example

```
gzip            on;
gzip_min_length 1000;
gzip_proxied    expired no-cache no-store private auth;
gzip_types      text/plain application/xml;
```

The *$gzip_ratio* variable can be used to log the achieved compression ratio.

Directives

gzip

Syntax	`gzip on | off;`
Default	`gzip off;`
Context	http, server, location, if in location

Enables or disables gzipping of responses.

gzip_buffers

Syntax	`gzip_buffers` *number size*;
Default	`gzip_buffers 32 4k | 16 8k;`
Context	http, server, location

Sets the number and size of buffers used to compress a response. By default, the buffer size is equal to one memory page. This is either 4K or 8K, depending on a platform.

gzip_comp_level

Syntax	`gzip_comp_level` *level*;
Default	`gzip_comp_level 1;`
Context	http, server, location

Sets a gzip compression level of a response. Acceptable values are in the range from 1 to 9.

gzip_disable

Syntax	`gzip_disable` *regex* ...;
Default	—
Context	http, server, location

Disables gzipping of responses for requests with "User-Agent" header fields matching any of the specified regular expressions.

The special mask `msie6` corresponds to the regular expression "*MSIE [4-6].*", but works faster. "*MSIE 6.0; ... SV1*" is excluded from this mask.

gzip_http_version

Syntax	`gzip_http_version 1.0 | 1.1;`
Default	`gzip_http_version 1.1;`
Context	http, server, location

Sets the minimum HTTP version of a request required to compress a response.

gzip_min_length

Syntax	`gzip_min_length` *length*;
Default	`gzip_min_length 20;`
Context	http, server, location

Sets the minimum length of a response that will be gzipped. The length is determined only from the "Content-Length" response header field.

gzip_proxied

Syntax	`gzip_proxied off | expired | no-cache | no-store | private | no_last_modified | no_etag | auth | any` ...;
Default	`gzip_proxied off;`
Context	http, server, location

Enables or disables gzipping of responses for proxied requests depending on the request and response. The fact that the request is proxied is determined by the presence of the "Via" request header field. The directive accepts multiple parameters:

`off`	disables compression for all proxied requests, ignoring other parameters;
`expired`	enables compression if a response header includes the "Expires" field with a value that disables caching;
`no-cache`	enables compression if a response header includes the "Cache-Control" field with the "no-cache" parameter;
`no-store`	enables compression if a response header includes the "Cache-Control" field with the "no-store" parameter;
`private`	enables compression if a response header includes the "Cache-Control" field with the "private" parameter;
`no_last_modified`	enables compression if a response header does not include the "Last-Modified" field;
`no_etag`	enables compression if a response header does not include the "ETag" field;
`auth`	enables compression if a request header includes the "Authorization" field;
`any`	enables compression for all proxied requests.

gzip_types

Syntax	`gzip_types` mime-*type* ...;
Default	`gzip_types text/html;`
Context	http, server, location

Enables gzipping of responses for the specified MIME types in addition to `text/html`. The special value "*" matches any MIME type. Responses with the `text/html` type are always compressed.

gzip_vary

Syntax	`gzip_vary on` | `off`;
Default	`gzip_vary off;`
Context	http, server, location

Enables or disables inserting the "Vary: Accept-Encoding" response header field if the directives *gzip*, *gzip_static* or *gunzip* are active.

Built-in Variables

`$gzip_ratio`

achieved compression ratio, computed as the ratio between the original and compressed response sizes.

GZip Static

Allows sending precompressed files with the ".gz" filename extension instead of regular files.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-http_gzip_static_module` build option.

In packages and images from our repos, the module is included in the build.

Configuration Example

```
gzip_static  on;
gzip_proxied expired no-cache no-store private auth;
```

Directives

gzip_static

Syntax	`gzip_static on | off | always;`
Default	`gzip_static off;`
Context	http, server, location

Enables (`on`) or disables (`off`) checking the existence of precompressed files. The following directives are also taken into account: *gzip_http_version*, *gzip_proxied*, *gzip_disable* and *gzip_vary*.

With `always`, gzipped files are used in all cases, without checking if the client supports it. This is useful if there are no uncompressed files on the disk anyway or the *GunZIP* module is used.

The files can be compressed using the gzip command, or any other compatible one. It is recommended that the modification date and time of original and compressed files be the same.

Headers

Allows adding the "Expires" and "Cache-Control" header fields, and arbitrary fields, to a response header.

Configuration Example

```
expires    24h;
expires    modified +24h;
expires    @24h;
expires    0;
expires    -1;
expires    epoch;
expires    $expires;
add_header Cache-Control private;
```

Directives

add_header

Syntax	`add_header` *name value* [`always`];
Default	—
Context	http, server, location, if in location

Adds the specified field to a response header provided that the response code equals 200, 201, 204, 206, 301, 302, 303, 304, 307, or 308. Parameter value can contain variables.

There could be several `add_header` directives. These directives are inherited from the previous configuration level if and only if there are no `add_header` directives defined on the current level.

If the `always` parameter is specified, the header field will be added regardless of the response code.

add_trailer

Syntax	`add_trailer` *name value* [`always`];
Default	—
Context	http, server, location, if in location

Adds the specified field to the end of a response provided that the response code equals 200, 201, 206, 301, 302, 303, 307, or 308. Parameter value can contain variables.

There could be several `add_trailer` directives. These directives are inherited from the previous configuration level if and only if there are no `add_trailer` directives defined on the current level.

If the `always` parameter is specified the specified field will be added regardless of the response code.

expires

Syntax	`expires` [`modified`] *time*; `expires epoch` | `max` | `off`;
Default	`expires off;`
Context	http, server, location, if in location

Enables or disables adding or modifying the "Expires" and "Cache-Control" response header fields provided that the response code equals 200, 201, 204, 206, 301, 302, 303, 304, 307, or 308. The parameter can be a positive or negative *time*.

The time in the "Expires" field is computed as a sum of the current time and time specified in the directive. If the `modified` parameter is used then the time is computed as a sum of the file's modification time and the time specified in the directive.

In addition, it is possible to specify a time of day using the "@" prefix:

```
expires @15h30m;
```

The contents of the "Cache-Control" field depends on the sign of the specified time:

- time is negative — "Cache-Control: no-cache".
- time is positive or zero — "Cache-Control: max-age=`t`", where *t* is a time specified in the directive, in seconds.

`epoch`	sets "Expires" to the value "Thu, 01 Jan 1970 00:00:01 GMT", and "Cache-Control" to "no-cache".
`max`	sets "Expires" to the value "Thu, 31 Dec 2037 23:55:55 GMT", and "Cache-Control" to 10 years.
`off`	disables adding or modifying the "Expires" and "Cache-Control" response header fields.

The last parameter value can contain variables:

```
map $sent_http_content_type $expires {
    default          off;
    application/pdf 42d;
    ~image/          max;
}

expires $expires;
```

Image Filter

The module is a filter that transforms images in JPEG, GIF, PNG, and WebP formats.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-http_image_filter_module` build option.

In our repositories, the module is built dynamically and is available as a separate package named `angie-module-image-filter` or `angie-pro-module-image-filter`.

Important

This module utilizes the libgd library. It is recommended to use the latest available version of the library.

To transform images in WebP format, the `libgd` library must be compiled with WebP support.

Configuration Example

```
location /img/ {
    proxy_pass   http://backend;
    image_filter resize 150 100;
    image_filter rotate 90;
    error_page   415 = /empty;
}

location = /empty {
    empty_gif;
}
```

Directives

image_filter

Syntax	• `image_filter off`; • `image_filter test`; • `image_filter size`; • `image_filter` rotate 90 | 180 | 270; • `image_filter` resize *width height*; • `image_filter` crop *width height*;
Default	`image_filter off;`
Context	location

Sets the type of transformation to perform on images:

`off`	turns off module processing in a surrounding location.
`test`	ensures that responses are images in either JPEG, GIF, PNG, or WebP format. Otherwise, the 415 (Unsupported Media Type) error is returned.
`size`	outputs information about images in a JSON format, e.g.: *`"img" : { "width": 100, "height": 100, "type": "gif" }`* In case of an error, the output is as follows: `{}`
`rotate` 90|180|270	rotates images counter-clockwise by the specified number of degrees. Parameter value can contain variables. This mode can be used either alone or along with the `resize` and `crop` transformations.
`resize` *width height*	proportionally reduces an image to the specified sizes. To reduce by only one dimension, another dimension can be specified as "-". In case of an error, the server will return code 415 (Unsupported Media Type). Parameter values can contain variables. When used along with the *rotate* parameter, the rotation happens **after** reduction.
`crop` *width height*	proportionally reduces an image to the larger side size and crops extraneous edges by another side. To reduce by only one dimension, another dimension can be specified as "-". In case of an error, the server will return code 415 (Unsupported Media Type). Parameter values can contain variables. When used along with the *rotate* parameter, the rotation happens **before** reduction.

image_filter_buffer

Syntax	`image_filter_buffer` *size*;
Default	`image_filter_buffer 1M;`
Context	http, server, location

Sets the maximum size of the buffer used for reading images. When the size is exceeded the server returns error 415 (Unsupported Media Type).

image_filter_interlace

Syntax	`image_filter_interlace on` | `off`;
Default	`image_filter_interlace off;`
Context	http, server, location

If enabled, final images will be interlaced. For JPEG, final images will be in "progressive JPEG" format.

image_filter_jpeg_quality

Syntax	`image_filter_jpeg_quality` *quality*;
Default	`image_filter_jpeg_quality 75;`
Context	http, server, location

Sets the desired quality of the transformed JPEG images. Acceptable values are in the range from 1 to 100. Lesser values usually imply both lower image quality and less data to transfer. The maximum recommended value is 95. Parameter value can contain variables.

image_filter_sharpen

Syntax	`image_filter_sharpen` *percent*;
Default	`image_filter_sharpen 0;`
Context	http, server, location

Increases sharpness of the final image. The sharpness percentage can exceed 100. The `0` value disables sharpening. Parameter value can contain variables.

image_filter_transparency

Syntax	`image_filter_transparency on` | `off`;
Default	`image_filter_transparency on;`
Context	http, server, location

Defines whether transparency should be preserved when transforming GIF images or PNG images with colors specified by a palette. The loss of transparency results in images of a better quality. The alpha channel transparency in PNG is always preserved.

image_filter_webp_quality

Syntax	`image_filter_webp_quality` *quality*;
Default	`image_filter_webp_quality 80;`
Context	http, server, location

Sets the desired quality of the transformed WebP images. Acceptable values are in the range from 1 to 100. Lesser values usually imply both lower image quality and less data to transfer. Parameter value can contain variables.

Index

The module processes requests ending with the slash character (`/`). Such requests can also be processed by the *http_autoindex* and *http_random_index* modules.

Configuration Example

```
location / {
    index index.$geo.html index.html;
}
```

Directives

index

Syntax	`index` *file* ...;
Default	`index index.html;`
Context	http, server, location

Defines files that will be used as an index. The file name can contain variables. Files are checked in the specified order. The last element of the list can be a file with an absolute path. Example:

```
index index.$geo.html index.0.html /index.html;
```

It should be noted that using an index file causes an internal redirect, and the request can be processed in a different location. For example, with the following configuration:

```
location = / {
    index index.html;
}

location / {
#    ...
}
```

A "" request will actually be processed in the second location as `/index.html`.

JS

The module is used to implement handlers in njs — a subset of the JavaScript language.

In our repositories, the module is built dynamically and is available as a separate package named `angie-module-njs` or `angie-pro-module-njs`.

Configuration Example

```
http {
    js_import http.js;

    js_set $foo     http.foo;
    js_set $summary http.summary;
    js_set $hash    http.hash;
```

```
    resolver 127.0.0.53;

    server {
        listen 8000;

        location / {
            add_header X-Foo $foo;
            js_content http.baz;
        }

        location = /summary {
            return 200 $summary;
        }

        location = /hello {
            js_content http.hello;
        }

        location = /fetch {
            js_content                    http.fetch;
            js_fetch_trusted_certificate /path/to/ISRG_Root_X1.pem;
        }

        location = /crypto {
            add_header Hash $hash;
            return     200;
        }
    }
}
```

The `http.js` file:

```
function foo(r) {
    r.log("hello from foo() handler");
    return "foo";
}

function summary(r) {
    var a, s, h;

    s = "JS summary\n\n";

    s += "Method: " + r.method + "\n";
    s += "HTTP version: " + r.httpVersion + "\n";
    s += "Host: " + r.headersIn.host + "\n";
    s += "Remote Address: " + r.remoteAddress + "\n";
    s += "URI: " + r.uri + "\n";

    s += "Headers:\n";
    for (h in r.headersIn) {
        s += "  header '" + h + "' is '" + r.headersIn[h] + "'\n";
    }

    s += "Args:\n";
    for (a in r.args) {
        s += "  arg '" + a + "' is '" + r.args[a] + "'\n";
    }
```

```
    return s;
}

function baz(r) {
    r.status = 200;
    r.headersOut.foo = 1234;
    r.headersOut['Content-Type'] = "text/plain; charset=utf-8";
    r.headersOut['Content-Length'] = 15;
    r.sendHeader();
    r.send("nginx");
    r.send("java");
    r.send("script");

    r.finish();
}

function hello(r) {
    r.return(200, "Hello world!");
}

async function fetch(r) {
    let results = await Promise.all([ngx.fetch('https://google.com/'),
                                     ngx.fetch('https://google.ru/')]);

    r.return(200, JSON.stringify(results, undefined, 4));
}

async function hash(r) {
    let hash = await crypto.subtle.digest('SHA-512', r.headersIn.host);
    r.setReturnValue(Buffer.from(hash).toString('hex'));
}

export default {foo, summary, baz, hello, fetch, hash};
```

Directives

js_body_filter

Syntax	`js_body_filter` *function* | *module.function* [`buffer_type=`*string* | *buffer*];
Default	—
Context	location, if in location, limit_except

Sets an njs function as a response body filter. The filter function is called for each data chunk of a response body with the following arguments:

`r`	the HTTP request object
`data`	the incoming data chunk, may be a string or Buffer depending on the *buffer_type* value, by default is a string.
`flags`	an object with the following properties: - `last` — a boolean value, `true` if data is the last buffer

The filter function can pass its own modified version of the input data chunk to the next body filter by calling r.sendBuffer(). For example, to transform all the lowercase letters in the response body:

```
function filter(r, data, flags) {
    r.sendBuffer(data.toLowerCase(), flags);
}
```

To stop filtering (following data chunks will be passed to client without calling *js_body_filter*), r.done() can be used.

If the filter function changes the length of the response body, then it is required to clear out the "Content-Length" response header (if any) in *js_header_filter* to enforce chunked transfer encoding.

Note

As the *js_body_filter* handler returns its result immediately, it supports only synchronous operations. Thus, asynchronous operations such as r.subrequest() or setTimeout() are not supported.

js_content

Syntax	`js_content` *function* | *module.function*;
Default	—
Context	location, if in location, limit_except

Sets an njs function as a location content handler. Module functions can be referenced.

js_fetch_buffer_size

Syntax	`js_fetch_buffer_size` *size*;
Default	`js_fetch_buffer_size 16k;`
Context	http, server, location

Sets the size of the buffer used for reading and writing with Fetch API.

js_fetch_ciphers

Syntax	`js_fetch_ciphers` *ciphers*;
Default	`js_fetch_ciphers HIGH:!aNULL:!MD5;`
Context	http, server, location

Specifies the enabled ciphers for HTTPS connections with Fetch API. The ciphers are specified in the format understood by the OpenSSL library.

The list of ciphers depends on the version of OpenSSL installed. The full list can be viewed using the `openssl ciphers` command.

js_fetch_max_response_buffer_size

Syntax	`js_fetch_max_response_buffer_size` *size*;
Default	`js_fetch_max_response_buffer_size 1m;`
Context	http, server, location

Sets the maximum size of the response received with Fetch API.

js_fetch_protocols

Syntax	`js_fetch_protocols` [TLSv1] [TLSv1.1] [TLSv1.2] [TLSv1.3];
Default	`js_fetch_protocols TLSv1 TLSv1.1 TLSv1.2;`
Context	http, server, location

Enables the specified protocols for HTTPS connections with Fetch API.

js_fetch_timeout

Syntax	`js_fetch_timeout` *time*;
Default	`js_fetch_timeout 60s;`
Context	http, server, location

Defines a timeout for reading and writing for Fetch API. The timeout is set only between two successive read/write operations, not for the whole response. If no data is transmitted within this time, the connection is closed.

js_fetch_trusted_certificate

Syntax	`js_fetch_trusted_certificate` *file*;
Default	—
Context	http, server, location

Specifies a file with trusted CA certificates in the PEM format used to verify the HTTPS certificate with Fetch API.

js_fetch_verify

Syntax	`js_fetch_verify on` | `off`;
Default	`js_fetch_verify on;`
Context	http, server, location

Enables or disables verification of the HTTPS server certificate with Fetch API.

js_fetch_verify_depth

Syntax	`js_fetch_verify_depth` *number*;
Default	`js_fetch_verify_depth 100;`
Context	http, server, location

Sets the verification depth in the HTTPS server certificates chain with Fetch API.

js_header_filter

Syntax	`js_header_filter` *function* | *module.function*;
Default	—
Context	location, if in location, limit_except

Sets an njs function as a response header filter. The directive allows changing arbitrary header fields of a response header.

> **Note**
>
> As the *js_header_filter* handler returns its result immediately, it supports only synchronous operations. Thus, asynchronous operations such as r.subrequest() or setTimeout() are not supported.

js_import

Syntax	`js_import` *module.js* | *export_name from module.js*;
Default	—
Context	http, server, location

Imports a module that implements location and variable handlers in njs. The *export_name* is used as a namespace to access module functions. If the *export_name* is not specified, the module name will be used as a namespace.

```
js_import http.js;
```

Here, the module name `http` is used as a namespace while accessing exports. If the imported module exports `foo()`, `http.foo` is used to refer to it.

Several `js_import` directives can be specified.

js_path

Syntax	`js_path` *path*;
Default	—
Context	http, server, location

Sets an additional path for njs modules.

js_preload_object

Syntax	`js_preload_object` *name.json* | *name from file.json*;
Default	—
Context	http, server, location

Preloads an immutable object at configure time. The *name* is used as a name of the global variable though which the object is available in njs code. If the *name* is not specified, the file name will be used instead.

```
js_preload_object map.json;
```

Here, the *map* is used as a name while accessing the preloaded object.

Several *js_preload_object* directives can be specified.

js_set

Syntax	`js_set` *$variable function* | *module.function*;
Default	—
Context	http, server, locatio

Sets an njs function for the specified variable. Module functions can be referenced.

The function is called when the variable is referenced for the first time for a given request. The exact moment depends on a *phase* at which the variable is referenced. This can be used to perform some logic not related to variable evaluation. For example, if the variable is referenced only in the *log_format* directive, its handler will not be executed until the log phase. This handler can be used to do some cleanup right before the request is freed.

Note

As the *js_set* handler returns its result immediately, it supports only synchronous callbacks. Thus, asynchronous callbacks such as r.subrequest() or setTimeout() are not supported.

js_shared_dict_zone

Syntax	`js_shared_dict_zone zone=`*name:size* [`timeout=`*time*] [`type=`*string* | *number*] [`evict`];
Default	—
Context	http

Sets the name and size of the shared memory zone that keeps the key-value dictionary shared between worker processes.

`type`	optional parameter, allows redefining the value type to `number`; by default, the shared dictionary uses `string` for keys and values
`timeout`	optional parameter, sets the time after which all shared dictionary entries are removed from the zone
`evict`	optional parameter, removes the oldest key-value pair when the zone storage is exhausted

Examples:

```
example.conf:
    # Creates a 1Mb dictionary with string values,
    # removes key-value pairs after 60 seconds of inactivity:
    js_shared_dict_zone zone=foo:1M timeout=60s;

    # Creates a 512Kb dictionary with string values,
    # forcibly removes oldest key-value pairs when the zone is exhausted:
    js_shared_dict_zone zone=bar:512K timeout=30s evict;

    # Creates a 32Kb permanent dictionary with number values:
    js_shared_dict_zone zone=num:32k type=number;
```

```
example.js:
    function get(r) {
        r.return(200, ngx.shared.foo.get(r.args.key));
```

```
    }

    function set(r) {
        r.return(200, ngx.shared.foo.set(r.args.key, r.args.value));
    }

    function delete(r) {
        r.return(200, ngx.shared.bar.delete(r.args.key));
    }

    function increment(r) {
        r.return(200, ngx.shared.num.incr(r.args.key, 2));
    }
```

js_var

Syntax	`js_var` *$variable* [*value*];
Default	—
Context	stream, server

Declares a writable variable. The value can contain text, variables, and their combination. The variable is not overwritten after a redirect unlike variables created with the *set* directive.

Request Argument

Each HTTP njs handler receives one argument, a request object.

Limit Conn

The module is used to limit the number of connections per the defined key, in particular, the number of connections from a single IP address.

Not all connections are counted. A connection is counted only if it has a request being processed by the server and the whole request header has already been read.

Configuration Example

```
http {
    limit_conn_zone $binary_remote_addr zone=addr:10m;

    ...

    server {

        ...

        location /download/ {
            limit_conn addr 1;
        }
```

Directives

limit_conn

Syntax	`limit_conn` *zone number*;
Default	—
Context	http, server, location

Sets the shared memory zone and the maximum allowed number of connections for a given key value. When this limit is exceeded, the server will return the *error* in reply to a request. For example, the directives

```
limit_conn_zone $binary_remote_addr zone=addr:10m;

server {
    location /download/ {
        limit_conn addr 1;
    }
```

allow only one connection per IP address at a time.

Note

In HTTP/2 and SPDY, each concurrent request is considered a separate connection.

There could be several `limit_conn` directives. For example, the following configuration will limit the number of connections to the server per client IP and, at the same time, the total number of connections to the virtual server:

```
limit_conn_zone $binary_remote_addr zone=perip:10m;
limit_conn_zone $server_name zone=perserver:10m;

server {
    ...
    limit_conn perip 10;
    limit_conn perserver 100;
}
```

These directives are inherited from the previous configuration level if and only if there are no `limit_conn` directives defined on the current level.

limit_conn_dry_run

Syntax	`limit_conn_dry_run on` | `off`;
Default	`limit_conn_dry_run off;`
Context	http, server, location

Enables the dry run mode. In this mode, the number of connections is not limited, however, in the *shared memory zone*, the number of excessive connections is accounted as usual.

limit_conn_log_level

Syntax	`limit_conn_log_level info` | `notice` | `warn` | `error`;
Default	`limit_conn_log_level error;`
Context	http, server, location

Sets the desired logging level for cases when the server limits the number of connections.

limit_conn_status

Syntax	`limit_conn_status` *code*;
Default	`limit_conn_status 503;`
Context	http, server, location

Sets the status code to return in response to rejected requests.

limit_conn_zone

Syntax	`limit_conn_zone` *key* zone = *name:size*;
Default	—
Context	http

Sets parameters for a shared memory zone that will keep states for various keys. In particular, the state includes the current number of connections. The key can contain text, variables, and their combination. Requests with an empty key value are not accounted.

Usage example:

```
limit_conn_zone $binary_remote_addr zone=addr:10m;
```

Here, a client IP address serves as a key. Note that instead of `$remote_addr`, the `$binary_remote_addr` variable is used here.

The `$remote_addr` variable's size can vary from 7 to 15 bytes. The stored state occupies either 32 or 64 bytes of memory on 32-bit platforms and always 64 bytes on 64-bit platforms.

The `$binary_remote_addr` variable's size is always 4 bytes for IPv4 addresses or 16 bytes for IPv6 addresses. The stored state always occupies 32 or 64 bytes on 32-bit platforms and 64 bytes on 64-bit platforms.

One megabyte zone can keep about 32 thousand 32-byte states or about 16 thousand 64-byte states. If the zone storage is exhausted, the server will return the *error* to all further requests.

Built-in Variables

`$limit_conn_status`

keeps the result of limiting the number of connections: `PASSED`, `REJECTED` or `REJECTED_DRY_RUN`

Limit Req

The module is used to limit the request processing rate per a defined key, in particular, the processing rate of requests coming from a single IP address. The limitation is done using the "leaky bucket" method.

Configuration Example

```
http {
    limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;

    ...

    server {
```

```
        ...

        location /search/ {
            limit_req zone=one burst=5;
        }
```

Directives

limit_req

Syntax	`limit_req` *zone number* [`burst=`*number*] [nodelay | `delay=`*number*];
Default	—
Context	http, server, location

Sets the shared memory zone and the maximum burst size of requests. If the requests rate exceeds the rate configured for a zone, their processing is delayed such that requests are processed at a defined rate. Excessive requests are delayed until their number exceeds the maximum burst size in which case the request is terminated with an *error*. By default, the maximum burst size is equal to zero. For example, the directives

```
limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;

server {
    location /search/ {
        limit_req zone=one burst=5;
    }
```

allow not more than 1 request per second at an average, with bursts not exceeding 5 requests.

If delaying of excessive requests while requests are being limited is not desired, the parameter `nodelay` should be used:

```
limit_req zone=one burst=5 nodelay;
```

The `delay` parameter specifies a limit at which excessive requests become delayed. Default value is zero, i.e. all excessive requests are delayed.

There could be several `limit_req` directives. For example, the following configuration will limit the processing rate of requests coming from a single IP address and, at the same time, the request processing rate by the virtual server:

```
limit_req_zone $binary_remote_addr zone=perip:10m rate=1r/s;
limit_req_zone $server_name zone=perserver:10m rate=10r/s;

server {
    ...
    limit_req zone=perip burst=5 nodelay;
    limit_req zone=perserver burst=10;
}
```

These directives are inherited from the previous configuration level if and only if there are no `limit_req` directives defined on the current level.

limit_req_dry_run

Syntax	`limit_req_dry_run on | off;`
Default	`limit_req_dry_run off;`
Context	http, server, location

Enables the dry run mode. In this mode, requests processing rate is not limited, however, in the *shared memory zone*, the number of excessive requests is accounted as usual.

limit_req_log_level

Syntax	`limit_req_log_level info | notice | warn | error;`
Default	`limit_req_log_level error;`
Context	http, server, location

Sets the desired logging level for cases when the server refuses to process requests due to rate exceeding, or delays request processing. Logging level for delays is one point less than for refusals; for example, if `limit_req_log_level notice` is specified, delays are logged with the `info` level.

limit_req_status

Syntax	`limit_req_status` *code*;
Default	`limit_req_status 503;`
Context	http, server, location

Sets the status code to return in response to rejected requests.

limit_req_zone

Syntax	`limit_req_zone` *key* `zone=`*name:size* `rate=`*rate*;
Default	—
Context	http

Sets parameters for a shared memory zone that will keep states for various keys. In particular, the state stores the current number of excessive requests. The key can contain text, variables, and their combination. Requests with an empty key value are not accounted.

Usage example:

```
limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;
```

Here, the states are kept in a 10 megabyte zone `one`, and an average request processing rate for this zone cannot exceed 1 request per second.

A client IP address serves as a key. Note that instead of `$remote_addr`, the `$binary_remote_addr` variable is used here.

The `$binary_remote_addr` variable's size is always 4 bytes for IPv4 addresses or 16 bytes for IPv6 addresses. The stored state always occupies 64 bytes on 32-bit platforms and 128 bytes on 64-bit platforms.

One megabyte zone can keep about 16 thousand 64-byte states or about 8 thousand 128-byte states.

If the zone storage is exhausted, the least recently used state is removed. If even after that a new state cannot be created, the request is terminated with an *error*.

The `rate` is specified in requests per second (r/s). If a rate of less than one request per second is desired, it is specified in request per minute (r/m). For example, half-request per second is 30r/m.

Built-in Variables

`$limit_req_status`

keeps the result of limiting the request processing rate: `PASSED`, `DELAYED`, `REJECTED`, `DELAYED_DRY_RUN`, or `REJECTED_DRY_RUN`

Log

The module writes request logs in the specified format.

Requests are logged in the context of a location where processing ends. It may be different from the original location, if an *internal redirect* happens during request processing.

Configuration Example

```
log_format compression '$remote_addr - $remote_user [$time_local] '
                       '"$request" $status $bytes_sent '
                       '"$http_referer" "$http_user_agent" "$gzip_ratio"';

access_log /spool/logs/angie-access.log compression buffer=32k;
```

Directives

access_log

Syntax	`access_log` *path* [*format* [`buffer=`*size*] [gzip[=*level*]] [`flush=`*time*] [`if=`*condition*]]; `access_log off;`
Default	`access_log logs/access.log combined;` (the path depends on the `--http-log-path` build option)
Context	http, server, location, if in location, limit_except

Sets the `path`, `format`, and configuration for a buffered log write. Several logs can be specified on the same configuration level. Logging to *syslog* can be configured by specifying the `"syslog:"` prefix in the first parameter. The special value `off` cancels all `access_log` directives on the current level. If the format is not specified then the predefined `"combined"` format is used.

If either the `buffer` or `gzip` parameter is used, writes to log will be buffered.

Caution

The buffer size must not exceed the size of an atomic write to a disk file. For FreeBSD this size is unlimited.

When buffering is enabled, the data will be written to the file:

- if the next log line does not fit into the buffer;
- if the buffered data is older than specified by the flush parameter;
- when a worker process is *re-opening log files* or is shutting down.

If the `gzip` parameter is used, then the buffered data will be compressed before writing to the file. The compression level can be set between `1` (fastest, less compression) and `9` (slowest, best compression). By

default, the buffer size is equal to `64K` bytes, and the compression level is set to `1`. Since the data is compressed in atomic blocks, the log file can be decompressed or read by `"zcat"` at any time.

Example:

```
access_log /path/to/log.gz combined gzip flush=5m;
```

Important

For gzip compression to work, Angie must be built with the zlib library.

The file path can contain variables, but such logs have some constraints:

- the *user* whose credentials are used by worker processes should have permissions to create files in a directory with such logs;
- buffered writes do not work;
- the file is opened and closed for each log write. However, since the descriptors of frequently used files can be stored in a cache, writing to the old file can continue during the time specified by the *open_log_file_cache* directive's `valid` parameter
- during each log write the existence of the request's root directory is checked, and if it does not exist the log is not created. It is thus a good idea to specify both *root* and *access_log* on the same configuration level:

```
server {
    root       /spool/vhost/data/$host;
    access_log /spool/vhost/logs/$host;
    ...
```

The `if` parameter enables conditional logging. A request will not be logged if the condition evaluates to `"0"` or an empty string. In the following example, the requests with response codes 2xx and 3xx will not be logged:

```
map $status $loggable {
    ~^[23]  0;
    default 1;
}

access_log /path/to/access.log combined if=$loggable;
```

log_format

Syntax	`log_format` *name* [`escape=default` | `json` | `none`] *string* ...;
Default	`log_format combined "...";`
Context	http

Specifies log format.

The `escape` parameter allows setting `json` or `default` characters escaping in variables, by default, `default` escaping is used. The none value disables escaping.

For `default` escaping, characters """, "\", and other characters with values less than 32 or above 126 are escaped as "\xXX". If the variable value is not found, a hyphen "-" will be logged.

For `json` escaping, all characters not allowed in JSON strings will be escaped: characters """ and "\" are escaped as "\"" and "\\", characters with values less than 32 are escaped as "\n", "\r", "\t", "\b", "\f", or "\u00XX".

Header lines sent to a client have the prefix `sent_http_`, for example, `$sent_http_content_range`.

The configuration always includes the predefined `combined` format:

```
log_format combined '$remote_addr - $remote_user [$time_local] '
                    '"$request" $status $body_bytes_sent '
                    '"$http_referer" "$http_user_agent"';
```

open_log_file_cache

Syntax	`open_log_file_cache` `max=`*N* [`inactive=`*time*] [`min_uses=`*N*] [`valid=`*time*]; `open_log_file_cache off;`
Default	`open_log_file_cache off;`
Context	http, server, location

Defines a cache that stores the file descriptors of frequently used logs whose names contain variables. The directive has the following parameters:

`max`	sets the maximum number of descriptors in a cache; if the cache becomes full the least recently used (LRU) descriptors are closed
`inactive`	sets the time after which the cached descriptor is closed if there were no access during this time; by default, 10 seconds
`min_uses`	sets the minimum number of file uses during the time defined by the `inactive` parameter to let the descriptor stay open in a cache; by default, 1
`valid`	sets the time after which it should be checked that the file still exists with the same name; by default, 60 seconds
`off`	disables caching

Usage example:

```
open_log_file_cache max=1000 inactive=20s valid=1m min_uses=2;
```

Map

The module creates variables whose values depend on values of other variables.

Configuration Example

```
map $http_host $name {
    hostnames;

    default       0;

    example.com   1;
    *.example.com 1;
    example.org   2;
    *.example.org 2;
    .example.net  3;
    wap.*         4;
}

map $http_user_agent $mobile {
    default       0;
    "~Opera Mini" 1;
}
```

Directives

map

Syntax	`map` *string $variable* { ... }
Default	—
Context	http

Creates a new variable. Its value depends on the first parameter, specified as a string with variables, for example:

```
set $var1 "foo";
set $var2 "bar";

map $var1$var2 $new_variable {
    default "foobar_value";
}
```

Here, the variable `$new_variable` will have a value composed of the two variables `$var1` and `$var2`, or a default value if these variables are not defined.

> **Note**
>
> Since variables are evaluated only when they are used, the mere declaration even of a large number of "map" variables does not add any extra costs to request processing.

Parameters inside the *map* block specify a mapping between source and resulting values.

Source values are specified as strings or regular expressions.

Strings are matched ignoring the case.

A regular expression should either start with a "~" symbol for a case-sensitive matching, or with the "~*" symbols for case-insensitive matching. A regular expression can contain named and positional captures that can later be used in other directives along with the resulting variable.

If a source value matches one of the names of special parameters described below, it should be prefixed with the "" symbol.

The resulting value can contain text, variable and their combination.

The following special parameters are also supported:

`default` *value*	sets the resulting value if the source value matches none of the specified variants. When *default* is not specified, the default resulting value will be an empty string.
`hostnames`	indicates that source values can be hostnames with a prefix or suffix mask. This parameter should be specified before the list of values.

For example,

```
*.example.com 1;
example.*     1;
```

The following two records

```
example.com   1;
*.example.com 1;
```

can be combined:

```
.example.com  1;
```

`include` *file*	includes a file with values. There can be several inclusions.
`volatile`	indicates that the variable is not cacheable.

If the source value matches more than one of the specified variants, e.g. both a mask and a regular expression match, the first matching variant will be chosen, in the following order of priority:

1. String value without a mask
2. Longest string value with a prefix mask, e.g. `*.example.com`
3. Longest string value with a suffix mask, e.g. `mail.*`
4. First matching regular expression (in order of appearance in a configuration file)
5. Default value (`default`)

map_hash_bucket_size

Syntax	`map_hash_bucket_size` *size*;
Default	`map_hash_bucket_size 32|64|128;`
Context	http

Sets the bucket size for the *map* variables hash tables. Default value depends on the processor's cache line size. The details of setting up hash tables are provided *separately*.

map_hash_max_size

Syntax	`map_hash_max_size` *size*;
Default	`map_hash_max_size 2048;`
Context	http

Sets the maximum size of the *map* variables hash tables. The details of setting up hash tables are provided *separately*.

Memcached

The module is used to obtain responses from a memcached server. The key is set in the *$memcached_key* variable. A response should be put in memcached in advance by means external to Angie.

Configuration Example

```
server {
    location / {
        set             $memcached_key "$uri?$args";
        memcached_pass host:11211;
        error_page      404 502 504 = @fallback;
    }

    location @fallback {
        proxy_pass      http://backend;
    }
}
```

Directives

memcached_bind

Syntax	`memcached_bind` *address* [`transparent`] | `off`;
Default	—
Context	http, server, location

Makes outgoing connections to a memcached server originate from the specified local IP address with an optional port. Parameter value can contain variables. The special value `off` cancels the effect of the *memcached_bind* directive inherited from the previous configuration level, which allows the system to auto-assign the local IP address and port.

The `transparent` parameter allows outgoing connections to a memcached server originate from a non-local IP address, for example, from a real IP address of a client:

```
memcached_bind $remote_addr transparent;
```

In order for this parameter to work, it is usually necessary to run Angie worker processes with the *superuser* privileges. On Linux it is not required as if the `transparent` parameter is specified, worker processes inherit the *CAP_NET_RAW* capability from the master process.

Important

It is necessary to configure kernel routing table to intercept network traffic from the memcached server.

memcached_buffer_size

Syntax	`memcached_buffer_size` *size*;
Default	`memcached_buffer_size 4k|8k;`
Context	http, server, location

Sets the size of the buffer used for reading the first part of the response received from the memcached server. The response is passed to the client synchronously, as soon as it is received.

memcached_connect_timeout

Syntax	`memcached_connect_timeout` *time*;
Default	`memcached_connect_timeout 60s;`
Context	http, server, location

Defines a timeout for establishing a connection with a memcached server. It should be noted that this timeout cannot usually exceed 75 seconds.

memcached_gzip_flag

Syntax	`memcached_gzip_flag` *flag*;
Default	—
Context	http, server, location

Enables the test for the flag presence in the memcached server response and sets the "Content-Encoding" response header field to "gzip" if the flag is set.

memcached_next_upstream

Syntax	`memcached_next_upstream error | timeout | invalid_response | not_found | off ...;`
Default	`memcached_next_upstream error timeout;`
Context	http, server, location

Specifies in which cases a request should be passed to the next server in the *upstream pool*:

`error`	an error occurred while establishing a connection with the server, passing a request to it, or reading the response header;
`timeout`	a timeout has occurred while establishing a connection with the server, passing a request to it, or reading the response header;
`invalid_response`	a server returned an empty or invalid response;
`not_found`	a response was not found on the server;
`off`	disables passing a request to the next server.

Note

One should bear in mind that passing a request to the next server is only possible if nothing has been sent to a client yet. That is, if an error or timeout occurs in the middle of the transferring of a response, fixing this is impossible.

The directive also defines what is considered an *unsuccessful attempt* of communication with a server.

`error`, `timeout`, `invalid_response`	always considered unsuccessful attempts, even if they are not specified in the directive
`not_found`	never considered unsuccessful attempts

Passing a request to the next server can be limited by the *number of tries* and by *time*.

memcached_next_upstream_timeout

Syntax	`memcached_next_upstream_timeout` *time*;
Default	`memcached_next_upstream_timeout 0;`
Context	http, server, location

Limits the time during which a request can be passed to the *next* server.

`0`	turns off this limitation

memcached_next_upstream_tries

Syntax	`memcached_next_upstream_tries` *number*;
Default	`memcached_next_upstream_tries 0;`
Context	http, server, location

Limits the number of possible tries for passing a request to the *next* server.

0	turns off this limitation

memcached_pass

Syntax	`memcached_pass` *uri*;
Default	—
Context	location, if in location

Sets the memcached server address. The address can be specified as a domain name or IP address, and a port:

```
memcached_pass localhost:11211;
```

or as a UNIX domain socket path:

```
memcached_pass unix:/tmp/memcached.socket;
```

If a domain name resolves to several addresses, all of them will be used in a round-robin fashion. In addition, an address can be specified as a *server group*. If a group is used, you cannot specify the port with it; instead, specify the port for each server within the group individually.

memcached_read_timeout

Syntax	`memcached_read_timeout` *time*;
Default	`memcached_read_timeout 60s;`
Context	http, server, location

Defines a timeout for reading a response from the memcached server. The timeout is set only between two successive read operations, not for the transmission of the whole response. If the memcached server does not transmit anything within this time, the connection is closed.

memcached_send_timeout

Syntax	`memcached_send_timeout` *time*;
Default	`memcached_send_timeout 60s;`
Context	http, server, location

Sets a timeout for transmitting a request to the memcached server. The timeout is set only between two successive write operations, not for the transmission of the whole request. If the memcached server does not receive anything within this time, the connection is closed.

memcached_socket_keepalive

Syntax	`memcached_socket_keepalive on` | `off`;
Default	`memcached_socket_keepalive off;`
Context	http, server, location

Configures the "TCP keepalive" behavior for outgoing connections to a memcached server.

""	By default, the operating system's settings are in effect for the socket.
on	The *SO_KEEPALIVE* socket option is turned on for the socket.

Built-in Variables

`$memcached_key`

Defines a key for obtaining response from a memcached server.

Mirror

The module implements mirroring of an original request by creating background mirror subrequests. Responses to mirror subrequests are ignored.

Configuration Example

```
location / {
    mirror /mirror;
    proxy_pass http://backend;
}

location = /mirror {
    internal;
    proxy_pass http://test_backend$request_uri;
}
```

Directives

mirror

Syntax	`mirror` *uri* | `off;`
Default	`mirror off;`
Context	http, server, location

Sets the URI to which an original request will be mirrored. Several mirrors can be specified on the same configuration level.

mirror_request_body

Syntax	`mirror_request_body on` | `off;`
Default	`mirror_request_body on;`
Context	http, server, location

Indicates whether the client request body is mirrored. When enabled, the client request body will be read prior to creating mirror subrequests. In this case, unbuffered client request body proxying set by the *proxy_request_buffering*, *fastcgi_request_buffering*, *scgi_request_buffering* and *uwsgi_request_buffering* directives will be disabled.

```
location / {
    mirror /mirror;
    mirror_request_body off;
    proxy_pass http://backend;
}

location = /mirror {
    internal;
    proxy_pass http://log_backend;
    proxy_pass_request_body off;
```

```
    proxy_set_header Content-Length "";
    proxy_set_header X-Original-URI $request_uri;
}
```

MP4

The module provides pseudo-streaming server-side support for MP4 files. Such files typically have the .mp4, .m4v, or .m4a filename extensions.

In packages and images from our repos, the module is included in the build.

Pseudo-streaming works in alliance with a compatible media player. The player sends an HTTP request to the server with the start time specified in the query string argument (named simply start and specified in seconds), and the server responds with the stream such that its start position corresponds to the requested time, for example:

```
http://example.com/elephants_dream.mp4?start=238.88
```

This allows performing a random seeking at any time, or starting playback in the middle of the timeline.

To support seeking, H.264-based formats store metadata in a so-called "moov atom". It is a part of the file that holds the index information for the whole file.

To start playback, the player first needs to read metadata. This is done by sending a special request with the `start=0` argument. A lot of encoding software insert the metadata at the end of the file. This is suboptimal for pseudo-streaming, because the player has to download the entire file before starting playback. If the metadata are located at the beginning of the file, it is enough for Angie to simply start sending back the file contents. If the metadata are located at the end of the file, Angie must read the entire file and prepare a new stream so that the metadata come before the media data. This involves some CPU, memory, and disk I/O overhead, so it is a good idea to prepare an original file for pseudo-streaming in advance, rather than having Angie do this on every such request.

The module also supports the `end` argument of an HTTP request which sets the end point of playback. The `end` argument can be specified with the start argument or separately:

```
http://example.com/elephants_dream.mp4?start=238.88&end=555.55
```

For a matching request with a non-zero `start` or `end` argument, Angie will read the metadata from the file, prepare the stream with the requested time range, and send it to the client. This has the same overhead as described above.

If the `start` argument points to a non-key video frame, the beginning of such video will be broken. To fix this issue, the video *can* be prepended with the key frame before `start` point and with all intermediate frames between them. These frames will be hidden from playback using an edit list.

If a matching request does not include the `start` and `end` arguments, there is no overhead, and the file is sent simply as a static resource. Some players also support byte-range requests, and thus do not require this module.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-http_mp4_module` build option.

Caution

If a third-party `mp4` module was previously used, it should be disabled.

A similar pseudo-streaming support for FLV files is provided by the *FLV*.

Configuration Example

```
set_real_ip_from  192.168.1.0/24;
set_real_ip_from  192.168.2.1;
set_real_ip_from  2001:0db8::/32;
real_ip_header    X-Forwarded-For;
real_ip_recursive on;
```

Directives

mp4

Syntax	`mp4;`
Default	—
Context	location

Turns on module processing in a surrounding location.

mp4_buffer_size

Syntax	`mp4_buffer_size` *size*;
Default	`mp4_buffer_size 512K;`
Context	http, server, location

Sets the initial size of the buffer used for processing MP4 files.

mp4_max_buffer_size

Syntax	`mp4_max_buffer_size` *size*;
Default	`mp4_max_buffer_size 10M;`
Context	http, server, location

During metadata processing, a larger buffer may become necessary. Its size cannot exceed the specified size, or else Angie will return the 500 (Internal Server Error) server error, and log the following message:

> "/some/movie/file.mp4" mp4 moov atom is too large: 12583268, you may want to increase mp4_max_buffer_size

mp4_limit_rate

Syntax	`mp4_limit_rate on` | `off` | *factor*;
Default	`mp4_limit_rate off;`
Context	http, server, location

Rate-limits the transfer of the requested MP4 file to the client. To calculate the limit, the *factor* is multiplied by the average bitrate of the file.

- The `off` value disables rate limiting.
- The `on` value sets a *factor* of `1.1`.
- The limit is applied after reaching the value set by *mp4_limit_rate_after*.

The requests are rate limited individually: if the client opens two connections, the resulting rate doubles. In this regard, consider using *limit_conn* and accompanying directives.

mp4_limit_rate_after

Syntax	`mp4_limit_rate_after` *time*;
Default	`mp4_limit_rate_after 60s;`
Context	http, server, location

Sets (in terms of *playback time*) the amount of mediadata transferred that triggers the rate limit set by *mp4_limit_rate*.

mp4_start_key_frame

Syntax	`mp4_start_key_frame on` | `off`;
Default	`mp4_start_key_frame off;`
Context	http, server, location

Forces output video to always start with a key video frame. If the start argument does not point to a key frame, initial frames are hidden using an mp4 edit list. Edit lists are supported by major players and browsers such as Chrome, Safari, QuickTime and ffmpeg, partially supported by Firefox.

Perl

The module is used to implement location and variable handlers in Perl and insert Perl calls into SSI.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-http_perl_module` build option.

In our repositories, the module is built dynamically and is available as a separate package named `angie-module-perl` or `angie-pro-module-perl`.

Important

This module requires Perl version 5.6.1 or higher. The C compiler should be compatible with the one used to build Perl.

Known Issues

The module is experimental, caveat emptor applies.

In order for Perl to recompile the modified modules during reconfiguration, it should be built with the `-Dusemultiplicity=yes` or `-Dusethreads=yes` parameters. Also, to make Perl leak less memory at run time, it should be built with the `-Dusemymalloc=no` parameter. To check the values of these parameters in an already built Perl (preferred values are specified in the example), run:

```
$ perl -V:usemultiplicity -V:usemymalloc
usemultiplicity='define';
usemymalloc='n';
```

Note that after rebuilding Perl with the new `-Dusemultiplicity=yes` or `-Dusethreads=yes` parameters, all binary Perl modules will have to be rebuilt as well — they will just stop working with the new Perl.

There is a possibility that the main process and then worker processes will grow in size after every reconfiguration. If the main process grows to an unacceptable size, the *live upgrade* procedure can be applied without changing the executable file.

While the Perl module is performing a long-running operation, such as resolving a domain name, connecting to another server, or querying a database, other requests assigned to the current worker process will not be processed. It is thus recommended to perform only such operations that have predictable and short execution time, such as accessing the local file system.

Configuration Example

```
http {

    perl_modules perl/lib;
    perl_require hello.pm;

    perl_set $msie6 '

        sub {
            my $r = shift;
            my $ua = $r->header_in("User-Agent");

            return "" if $ua =~ /Opera/;
            return "1" if $ua =~ / MSIE [6-9]\.\d+/;
            return "";
        }

    ';

    server {
        location / {
            perl hello::handler;
        }
    }
```

The *perl/lib/hello.pm* module:

```
package hello;

use nginx;

sub handler {
    my $r = shift;

    $r->send_http_header("text/html");
    return OK if $r->header_only;

    $r->print("hello!\n<br/>");

    if (-f $r->filename or -d _) {
        $r->print($r->uri, " exists!\n");
    }

    return OK;
}

1;
__END__
```

Directives

perl

Syntax	`perl` *module* :: *function* | 'sub { ... }';
Default	—
Context	location, limit_except

Sets a Perl handler for the given location.

perl_modules

Syntax	`perl_modules` *path*;
Default	—
Context	http

Sets an additional path for Perl modules.

perl_require

Syntax	`perl_require` *module*;
Default	—
Context	http

Defines the name of a module that will be loaded during each reconfiguration. Several *perl_require* directives can be present.

perl_set

Syntax	`perl_set` *$variable module* :: *function* | 'sub { ... }';
Default	—
Context	http

Installs a Perl handler for the specified variable.

Calling Perl from SSI

An SSI command calling Perl has the following format:

```
<!--# perl sub="module::function" arg="parameter1" arg="parameter2" ...
-->
```

The $r Request Object Methods

`$r->args`

Returns request arguments.

`$r->filename`

Returns a filename corresponding to the request URI.

`$r->has_request_body` **(*handler*)**

Returns 0 if there is no body in a request. If there is a body, the specified handler is set for the request and 1 is returned. After reading the request body, Angie will call the specified handler. Note that the handler function should be passed by reference. Example:

```
package hello;

use nginx;
```

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

    if ($r->request_method ne "POST") {
        return DECLINED;
    }

    if ($r->has_request_body(\&post)) {
        return OK;
    }

    return HTTP_BAD_REQUEST;
}

sub post {
    my $r = shift;

    $r->send_http_header;

    $r->print("request_body: \"", $r->request_body, "\"<br/>");
    $r->print("request_body_file: \"", $r->request_body_file, "\"<br/>\n");

    return OK;
}

1;

__END__
```

`$r->allow_ranges`

Enables the use of byte ranges when sending responses.

`$r->discard_request_body`

Instructs Angie to discard the request body.

`$r->header_in` (***field***)

Returns the value of the specified client request header field.

`$r->header_only`

Determines whether the whole response or only its header should be sent to the client.

`$r->header_out` (***field, value***)

Sets a value for the specified response header field.

`$r->internal_redirect` (***uri***)

Does an internal redirect to the specified uri. An actual redirect happens after the Perl handler execution is completed. Method accepts escaped URIs and supports redirections to *named locations*.

`$r->log_error` **(*errno, message*)**

Writes the specified message into the *error_log*. If *errno* is non-zero, an error code and its description will be appended to the message.

`$r->print` **(*text, ...*)**

Passes data to a client.

`$r->request_body`

Returns the client request body if it has not been written to a temporary file. To ensure that the client request body is in memory, its size should be limited by *client_max_body_size*, and a sufficient buffer size should be set using *client_body_buffer_size*.

`$r->request_body_file`

Returns the name of the file with the client request body. After the processing, the file should be removed. To always write a request body to a file, *client_body_in_file_only* should be enabled.

`$r->request_method`

Returns the client request HTTP method.

`$r->remote_addr`

Returns the client IP address.

`$r->flush`

Immediately sends data to the client.

`$r->sendfile` **(*name* [, *offset* [, *length*]])**

Sends the specified file content to the client. Optional parameters specify the initial offset and length of the data to be transmitted. The actual data transmission happens after the Perl handler has completed.

`$r->send_http_header` **([*type*])**

Sends the response header to the client. The optional type parameter sets the value of the "Content-Type" response header field. If the value is an empty string, the "Content-Type" header field will not be sent.

`$r->status` **(*code*)**

Sets a response code.

`$r->sleep` **(*milliseconds, handler*)**

Sets the specified handler and stops request processing for the specified time. In the meantime, Angie continues to process other requests. After the specified time has elapsed, Angie will call the installed handler. Note that the handler function should be passed by reference. In order to pass data between handlers, *$r->variable()* should be used. Example:

```
package hello;

use nginx;
```

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

    $r->discard_request_body;
    $r->variable("var", "OK");
    $r->sleep(1000, \&next);

    return OK;
}

sub next {
    my $r = shift;

    $r->send_http_header;
    $r->print($r->variable("var"));

    return OK;
}

1;

__END__
```

`$r->unescape` **(*text*)**

Decodes a text encoded in the "%XX" form.

`$r->uri`

Returns a request URI.

`$r->variable` **(*name* [, *value*])**

Returns or sets the value of the specified variable. Variables are local to each request.

Prometheus

Collects the Angie *metrics*, using configuration-defined templates, and returns the template-generated metrics in the Prometheus format.

Attention

To collect these metrics, enable a shared memory zone in appropriate contexts using:

- the `zone` directive in an *http_ upstream* or a *stream_ upstream*;
- the *status_ zone* directive;
- the `status_zone` parameter in the *resolver* directive.

Configuration Example

Three metrics that collect request statistics for a server's shared memory zones, combined into the `custom` template and exposed at `/p8s`:

```
http {
```

```
    prometheus_template custom {
        'angie_http_server_zones_requests_total{zone="$1"}' $p8s_value
            path=~^/http/server_zones/([^/]+)/requests/total$
            type=counter;

        'angie_http_server_zones_requests_processing{zone="$1"}' $p8s_value
            path=~^/http/server_zones/([^/]+)/requests/processing$
            type=gauge;

        'angie_http_server_zones_requests_discarded{zone="$1"}' $p8s_value
            path=~^/http/server_zones/([^/]+)/requests/discarded$
            type=counter;
    }

    # ...

    server {

        listen 80;

        location =/p8s {
            prometheus custom;
        }

        # ...

    }
}
```

Angie has a supplementary `prometheus_all.conf` file that defines a number of common metrics to be used as the `all` template:

File Contents (Angie)

```
prometheus_template all {

angie_connections_accepted $p8s_value
    path=/connections/accepted
    type=counter
    'help=The total number of accepted client connections.';

angie_connections_dropped $p8s_value
    path=/connections/dropped
    type=counter
    'help=The total number of dropped client connections.';

angie_connections_active $p8s_value
    path=/connections/active
    type=gauge
    'help=The current number of active client connections.';

angie_connections_idle $p8s_value
    path=/connections/idle
    type=gauge
    'help=The current number of idle client connections.';
```

```
'angie_slabs_pages_used{zone="$1"}' $p8s_value
    path=~^/slabs/([^/]+)/pages/used$
    type=gauge
    'help=The number of currently used memory pages in a slab zone.';

'angie_slabs_pages_free{zone="$1"}' $p8s_value
    path=~^/slabs/([^/]+)/pages/free$
    type=gauge
    'help=The number of currently free memory pages in a slab zone.';


'angie_slabs_pages_slots_used{zone="$1",size="$2"}' $p8s_value
    path=~^/slabs/([^/]+)/slots/([^/]+)/used$
    type=gauge
    'help=The number of currently used memory slots of a specific size in a slab zone.
↪';

'angie_slabs_pages_slots_free{zone="$1",size="$2"}' $p8s_value
    path=~^/slabs/([^/]+)/slots/([^/]+)/free$
    type=gauge
    'help=The number of currently free memory slots of a specific size in a slab zone.
↪';

'angie_slabs_pages_slots_reqs{zone="$1",size="$2"}' $p8s_value
    path=~^/slabs/([^/]+)/slots/([^/]+)/reqs$
    type=counter
    'help=The total number of attempts to allocate a memory slot of a specific size␣
↪in a slab zone.';

'angie_slabs_pages_slots_fails{zone="$1",size="$2"}' $p8s_value
    path=~^/slabs/([^/]+)/slots/([^/]+)/fails$
    type=counter
    'help=The number of unsuccessful attempts to allocate a memory slot of a specific␣
↪size in a slab zone.';


'angie_resolvers_queries{zone="$1",type="$2"}' $p8s_value
    path=~^/resolvers/([^/]+)/queries/([^/]+)$
    type=counter
    'help=The number of queries of a specific type to resolve in a resolver zone.';

'angie_resolvers_sent{zone="$1",type="$2"}' $p8s_value
    path=~^/resolvers/([^/]+)/sent/([^/]+)$
    type=counter
    'help=The number of sent DNS queries of a specific type to resolve in a resolver␣
↪zone.';

'angie_resolvers_responses{zone="$1",status="$2"}' $p8s_value
    path=~^/resolvers/([^/]+)/responses/([^/]+)$
    type=counter
    'help=The number of resolution results with a specific status in a resolver zone.
↪';


'angie_http_server_zones_ssl_handshaked{zone="$1"}' $p8s_value
    path=~^/http/server_zones/([^/]+)/ssl/handshaked$
    type=counter
```

```
    'help=The total number of successful SSL handshakes in an HTTP server zone.';

'angie_http_server_zones_ssl_reuses{zone="$1"}' $p8s_value
    path=~^/http/server_zones/([^/]+)/ssl/reuses$
    type=counter
    'help=The total number of session reuses during SSL handshakes in an HTTP server␣
↪zone.';

'angie_http_server_zones_ssl_timedout{zone="$1"}' $p8s_value
    path=~^/http/server_zones/([^/]+)/ssl/timedout$
    type=counter
    'help=The total number of timed-out SSL handshakes in an HTTP server zone.';

'angie_http_server_zones_ssl_failed{zone="$1"}' $p8s_value
    path=~^/http/server_zones/([^/]+)/ssl/failed$
    type=counter
    'help=The total number of failed SSL handshakes in an HTTP server zone.';


'angie_http_server_zones_requests_total{zone="$1"}' $p8s_value
    path=~^/http/server_zones/([^/]+)/requests/total$
    type=counter
    'help=The total number of client requests received in an HTTP server zone.';

'angie_http_server_zones_requests_processing{zone="$1"}' $p8s_value
    path=~^/http/server_zones/([^/]+)/requests/processing$
    type=gauge
    'help=The number of client requests currently being processed in an HTTP server␣
↪zone.';

'angie_http_server_zones_requests_discarded{zone="$1"}' $p8s_value
    path=~^/http/server_zones/([^/]+)/requests/discarded$
    type=counter
    'help=The total number of client requests completed in an HTTP server zone␣
↪without sending a response.';


'angie_http_server_zones_responses{zone="$1",code="$2"}' $p8s_value
    path=~^/http/server_zones/([^/]+)/responses/([^/]+)$
    type=counter
    'help=The number of responses with a specific status in an HTTP server zone.';


'angie_http_server_zones_data_received{zone="$1"}' $p8s_value
    path=~^/http/server_zones/([^/]+)/data/received$
    type=counter
    'help=The total number of bytes received from clients in an HTTP server zone.';

'angie_http_server_zones_data_sent{zone="$1"}' $p8s_value
    path=~^/http/server_zones/([^/]+)/data/sent$
    type=counter
    'help=The total number of bytes sent to clients in an HTTP server zone.';


'angie_http_location_zones_requests_total{zone="$1"}' $p8s_value
    path=~^/http/location_zones/([^/]+)/requests/total$
    type=counter
```

```
    'help=The total number of client requests in an HTTP location zone.';

'angie_http_location_zones_requests_discarded{zone="$1"}' $p8s_value
    path=~^/http/location_zones/([^/]+)/requests/discarded$
    type=counter
    'help=The total number of client requests completed in an HTTP location zone␣
↪without sending a response.';


'angie_http_location_zones_responses{zone="$1",code="$2"}' $p8s_value
    path=~^/http/location_zones/([^/]+)/responses/([^/]+)$
    type=counter
    'help=The number of responses with a specific status in an HTTP location zone.';


'angie_http_location_zones_data_received{zone="$1"}' $p8s_value
    path=~^/http/location_zones/([^/]+)/data/received$
    type=counter
    'help=The total number of bytes received from clients in an HTTP location zone.';

'angie_http_location_zones_data_sent{zone="$1"}' $p8s_value
    path=~^/http/location_zones/([^/]+)/data/sent$
    type=counter
    'help=The total number of bytes sent to clients in an HTTP location zone.';


'angie_http_upstreams_peers_state{upstream="$1",peer="$2"}' $p8st_all_ups_state
    path=~^/http/upstreams/([^/]+)/peers/([^/]+)/state$
    type=gauge
    'help=The current state of an upstream peer in "HTTP": 1 - up, 2 - down, 3 -␣
↪unavailable, or 4 - recovering.';


'angie_http_upstreams_peers_selected_current{upstream="$1",peer="$2"}' $p8s_value
    path=~^/http/upstreams/([^/]+)/peers/([^/]+)/selected/current$
    type=gauge
    'help=The number of requests currently being processed by an upstream peer in
↪"HTTP".';

'angie_http_upstreams_peers_selected_total{upstream="$1",peer="$2"}' $p8s_value
    path=~^/http/upstreams/([^/]+)/peers/([^/]+)/selected/total$
    type=counter
    'help=The total number of attempts to use an upstream peer in "HTTP".';


'angie_http_upstreams_peers_responses{upstream="$1",peer="$2",code="$3"}' $p8s_value
    path=~^/http/upstreams/([^/]+)/peers/([^/]+)/responses/([^/]+)$
    type=counter
    'help=The number of responses with a specific status received from an upstream␣
↪peer in "HTTP".';


'angie_http_upstreams_peers_data_sent{upstream="$1",peer="$2"}' $p8s_value
    path=~^/http/upstreams/([^/]+)/peers/([^/]+)/data/sent$
    type=counter
    'help=The total number of bytes sent to an upstream peer in "HTTP".';
```

```
'angie_http_upstreams_peers_data_received{upstream="$1",peer="$2"}' $p8s_value
    path=~^/http/upstreams/([^/]+)/peers/([^/]+)/data/received$
    type=counter
    'help=The total number of bytes received from an upstream peer in "HTTP".';


'angie_http_upstreams_peers_health_fails{upstream="$1",peer="$2"}' $p8s_value
    path=~^/http/upstreams/([^/]+)/peers/([^/]+)/health/fails$
    type=counter
    'help=The total number of unsuccessful attempts to communicate with an upstream␣
↪peer in "HTTP".';

'angie_http_upstreams_peers_health_unavailable{upstream="$1",peer="$2"}' $p8s_value
    path=~^/http/upstreams/([^/]+)/peers/([^/]+)/health/unavailable$
    type=counter
    'help=The number of times when an upstream peer in "HTTP" became "unavailable"␣
↪due to reaching the max_fails limit.';

'angie_http_upstreams_peers_health_downtime{upstream="$1",peer="$2"}' $p8s_value
    path=~^/http/upstreams/([^/]+)/peers/([^/]+)/health/downtime$
    type=counter
    'help=The total time (in milliseconds) that an upstream peer in "HTTP" was
↪"unavailable".';


'angie_http_upstreams_keepalive{upstream="$1"}' $p8s_value
    path=~^/http/upstreams/([^/]+)/keepalive$
    type=gauge
    'help=The number of currently cached keepalive connections for an HTTP upstream.';


'angie_http_caches_responses{zone="$1",status="$2"}' $p8s_value
    path=~^/http/caches/([^/]+)/([^/]+)/responses$
    type=counter
    'help=The total number of responses processed in an HTTP cache zone with a␣
↪specific cache status.';

'angie_http_caches_bytes{zone="$1",status="$2"}' $p8s_value
    path=~^/http/caches/([^/]+)/([^/]+)/bytes$
    type=counter
    'help=The total number of bytes processed in an HTTP cache zone with a specific␣
↪cache status.';

'angie_http_caches_responses_written{zone="$1",status="$2"}' $p8s_value
    path=~^/http/caches/([^/]+)/([^/]+)/responses_written$
    type=counter
    'help=The total number of responses written to an HTTP cache zone with a specific␣
↪cache status.';

'angie_http_caches_bytes_written{zone="$1",status="$2"}' $p8s_value
    path=~^/http/caches/([^/]+)/([^/]+)/bytes_written$
    type=counter
    'help=The total number of bytes written to an HTTP cache zone with a specific␣
↪cache status.';


'angie_http_caches_size{zone="$1"}' $p8s_value
```

```
    path=~^/http/caches/([^/]+)/size$
    type=gauge
    'help=The current size (in bytes) of cached responses in an HTTP cache zone.';


'angie_http_caches_shards_size{zone="$1",path="$2"}' $p8s_value
    path=~^/http/caches/([^/]+)/shards/([^/]+)/size$
    type=gauge
    'help=The current size (in bytes) of cached responses in a shard path of an HTTP␣
↪cache zone.';


'angie_http_limit_conns{zone="$1",status="$2"}' $p8s_value
    path=~^/http/limit_conns/([^/]+)/([^/]+)$
    type=counter
    'help=The number of requests processed by an HTTP limit_conn zone with a specific␣
↪result.';

'angie_http_limit_reqs{zone="$1",status="$2"}' $p8s_value
    path=~^/http/limit_reqs/([^/]+)/([^/]+)$
    type=counter
    'help=The number of requests processed by an HTTP limit_reqs zone with a specific␣
↪result.';


'angie_stream_server_zones_ssl_handshaked{zone="$1"}' $p8s_value
    path=~^/stream/server_zones/([^/]+)/ssl/handshaked$
    type=counter
    'help=The total number of successful SSL handshakes in a stream server zone.';

'angie_stream_server_zones_ssl_reuses{zone="$1"}' $p8s_value
    path=~^/stream/server_zones/([^/]+)/ssl/reuses$
    type=counter
    'help=The total number of session reuses during SSL handshakes in a stream server␣
↪zone.';

'angie_stream_server_zones_ssl_timedout{zone="$1"}' $p8s_value
    path=~^/stream/server_zones/([^/]+)/ssl/timedout$
    type=counter
    'help=The total number of timed-out SSL handshakes in a stream server zone.';

'angie_stream_server_zones_ssl_failed{zone="$1"}' $p8s_value
    path=~^/stream/server_zones/([^/]+)/ssl/failed$
    type=counter
    'help=The total number of failed SSL handshakes in a stream server zone.';


'angie_stream_server_zones_connections_total{zone="$1"}' $p8s_value
    path=~^/stream/server_zones/([^/]+)/connections/total$
    type=counter
    'help=The total number of client connections received in a stream server zone.';

'angie_stream_server_zones_connections_processing{zone="$1"}' $p8s_value
    path=~^/stream/server_zones/([^/]+)/connections/processing$
    type=gauge
    'help=The number of client connections currently being processed in a stream␣
↪server zone.';
```

```
'angie_stream_server_zones_connections_discarded{zone="$1"}' $p8s_value
    path=~^/stream/server_zones/([^/]+)/connections/discarded$
    type=counter
    'help=The total number of client connections completed in a stream server zone␣
↪without establishing a session.';

'angie_stream_server_zones_connections_passed{zone="$1"}' $p8s_value
    path=~^/stream/server_zones/([^/]+)/connections/passed$
    type=counter
    'help=The total number of client connections in a stream server zone passed for␣
↪handling to a different listening socket.';


'angie_stream_server_zones_sessions{zone="$1",status="$2"}' $p8s_value
    path=~^/stream/server_zones/([^/]+)/sessions/([^/]+)$
    type=counter
    'help=The number of sessions finished with a specific status in a stream server␣
↪zone.';


'angie_stream_server_zones_data_received{zone="$1"}' $p8s_value
    path=~^/stream/server_zones/([^/]+)/data/received$
    type=counter
    'help=The total number of bytes received from clients in a stream server zone.';

'angie_stream_server_zones_data_sent{zone="$1"}' $p8s_value
    path=~^/stream/server_zones/([^/]+)/data/sent$
    type=counter
    'help=The total number of bytes sent to clients in a stream server zone.';


'angie_stream_upstreams_peers_state{upstream="$1",peer="$2"}' $p8st_all_ups_state
    path=~^/stream/upstreams/([^/]+)/peers/([^/]+)/state$
    type=gauge
    'help=The current state of an upstream peer in "stream": 1 - up, 2 - down, 3 -␣
↪unavailable, or 4 - recovering.';


'angie_stream_upstreams_peers_selected_current{upstream="$1",peer="$2"}' $p8s_value
    path=~^/stream/upstreams/([^/]+)/peers/([^/]+)/selected/current$
    type=gauge
    'help=The number of sessions currently being processed by an upstream peer in
↪"stream".';

'angie_stream_upstreams_peers_selected_total{upstream="$1",peer="$2"}' $p8s_value
    path=~^/stream/upstreams/([^/]+)/peers/([^/]+)/selected/total$
    type=counter
    'help=The total number of attempts to use an upstream peer in "stream".';


'angie_stream_upstreams_peers_data_sent{upstream="$1",peer="$2"}' $p8s_value
    path=~^/stream/upstreams/([^/]+)/peers/([^/]+)/data/sent$
    type=counter
    'help=The total number of bytes sent to an upstream peer in "stream".';

'angie_stream_upstreams_peers_data_received{upstream="$1",peer="$2"}' $p8s_value
```

```
    path=~^/stream/upstreams/([^/]+)/peers/([^/]+)/data/received$
    type=counter
    'help=The total number of bytes received from an upstream peer in "stream".';


'angie_stream_upstreams_peers_health_fails{upstream="$1",peer="$2"}' $p8s_value
    path=~^/stream/upstreams/([^/]+)/peers/([^/]+)/health/fails$
    type=counter
    'help=The total number of unsuccessful attempts to communicate with an upstream␣
↪peer in "stream".';

'angie_stream_upstreams_peers_health_unavailable{upstream="$1",peer="$2"}' $p8s_value
    path=~^/stream/upstreams/([^/]+)/peers/([^/]+)/health/unavailable$
    type=counter
    'help=The number of times when an upstream peer in "stream" became "unavailable"␣
↪due to reaching the max_fails limit.';

'angie_stream_upstreams_peers_health_downtime{upstream="$1",peer="$2"}' $p8s_value
    path=~^/stream/upstreams/([^/]+)/peers/([^/]+)/health/downtime$
    type=counter
    'help=The total time (in milliseconds) that an upstream peer in "stream" was
↪"unavailable".';
}

map $p8s_value $p8st_all_ups_state {
    volatile;
    "up"           1;
    "down"         2;
    "unavailable"  3;
    "recovering"   4;
#    "unhealthy"    5;
#    "checking"     6;
#    "draining"     7;
    "busy"         8;
    default        0;
}
```

File Contents (Angie PRO)

```
prometheus_template all {

angie_connections_accepted $p8s_value
    path=/connections/accepted
    type=counter
    'help=The total number of accepted client connections.';

angie_connections_dropped $p8s_value
    path=/connections/dropped
    type=counter
    'help=The total number of dropped client connections.';

angie_connections_active $p8s_value
    path=/connections/active
    type=gauge
    'help=The current number of active client connections.';

angie_connections_idle $p8s_value
```

```
    path=/connections/idle
    type=gauge
    'help=The current number of idle client connections.';


'angie_slabs_pages_used{zone="$1"}' $p8s_value
    path=~^/slabs/([^/]+)/pages/used$
    type=gauge
    'help=The number of currently used memory pages in a slab zone.';

'angie_slabs_pages_free{zone="$1"}' $p8s_value
    path=~^/slabs/([^/]+)/pages/free$
    type=gauge
    'help=The number of currently free memory pages in a slab zone.';


'angie_slabs_pages_slots_used{zone="$1",size="$2"}' $p8s_value
    path=~^/slabs/([^/]+)/slots/([^/]+)/used$
    type=gauge
    'help=The number of currently used memory slots of a specific size in a slab zone.
↪';

'angie_slabs_pages_slots_free{zone="$1",size="$2"}' $p8s_value
    path=~^/slabs/([^/]+)/slots/([^/]+)/free$
    type=gauge
    'help=The number of currently free memory slots of a specific size in a slab zone.
↪';

'angie_slabs_pages_slots_reqs{zone="$1",size="$2"}' $p8s_value
    path=~^/slabs/([^/]+)/slots/([^/]+)/reqs$
    type=counter
    'help=The total number of attempts to allocate a memory slot of a specific size␣
↪in a slab zone.';

'angie_slabs_pages_slots_fails{zone="$1",size="$2"}' $p8s_value
    path=~^/slabs/([^/]+)/slots/([^/]+)/fails$
    type=counter
    'help=The number of unsuccessful attempts to allocate a memory slot of a specific␣
↪size in a slab zone.';


'angie_resolvers_queries{zone="$1",type="$2"}' $p8s_value
    path=~^/resolvers/([^/]+)/queries/([^/]+)$
    type=counter
    'help=The number of queries of a specific type to resolve in a resolver zone.';

'angie_resolvers_sent{zone="$1",type="$2"}' $p8s_value
    path=~^/resolvers/([^/]+)/sent/([^/]+)$
    type=counter
    'help=The number of sent DNS queries of a specific type to resolve in a resolver␣
↪zone.';

'angie_resolvers_responses{zone="$1",status="$2"}' $p8s_value
    path=~^/resolvers/([^/]+)/responses/([^/]+)$
    type=counter
    'help=The number of resolution results with a specific status in a resolver zone.
↪';
```

```
'angie_http_server_zones_ssl_handshaked{zone="$1"}' $p8s_value
    path=~^/http/server_zones/([^/]+)/ssl/handshaked$
    type=counter
    'help=The total number of successful SSL handshakes in an HTTP server zone.';

'angie_http_server_zones_ssl_reuses{zone="$1"}' $p8s_value
    path=~^/http/server_zones/([^/]+)/ssl/reuses$
    type=counter
    'help=The total number of session reuses during SSL handshakes in an HTTP server␣
↪zone.';

'angie_http_server_zones_ssl_timedout{zone="$1"}' $p8s_value
    path=~^/http/server_zones/([^/]+)/ssl/timedout$
    type=counter
    'help=The total number of timed-out SSL handshakes in an HTTP server zone.';

'angie_http_server_zones_ssl_failed{zone="$1"}' $p8s_value
    path=~^/http/server_zones/([^/]+)/ssl/failed$
    type=counter
    'help=The total number of failed SSL handshakes in an HTTP server zone.';


'angie_http_server_zones_requests_total{zone="$1"}' $p8s_value
    path=~^/http/server_zones/([^/]+)/requests/total$
    type=counter
    'help=The total number of client requests received in an HTTP server zone.';

'angie_http_server_zones_requests_processing{zone="$1"}' $p8s_value
    path=~^/http/server_zones/([^/]+)/requests/processing$
    type=gauge
    'help=The number of client requests currently being processed in an HTTP server␣
↪zone.';

'angie_http_server_zones_requests_discarded{zone="$1"}' $p8s_value
    path=~^/http/server_zones/([^/]+)/requests/discarded$
    type=counter
    'help=The total number of client requests completed in an HTTP server zone␣
↪without sending a response.';


'angie_http_server_zones_responses{zone="$1",code="$2"}' $p8s_value
    path=~^/http/server_zones/([^/]+)/responses/([^/]+)$
    type=counter
    'help=The number of responses with a specific status in an HTTP server zone.';


'angie_http_server_zones_data_received{zone="$1"}' $p8s_value
    path=~^/http/server_zones/([^/]+)/data/received$
    type=counter
    'help=The total number of bytes received from clients in an HTTP server zone.';

'angie_http_server_zones_data_sent{zone="$1"}' $p8s_value
    path=~^/http/server_zones/([^/]+)/data/sent$
    type=counter
    'help=The total number of bytes sent to clients in an HTTP server zone.';
```

```
'angie_http_location_zones_requests_total{zone="$1"}' $p8s_value
    path=~^/http/location_zones/([^/]+)/requests/total$
    type=counter
    'help=The total number of client requests in an HTTP location zone.';

'angie_http_location_zones_requests_discarded{zone="$1"}' $p8s_value
    path=~^/http/location_zones/([^/]+)/requests/discarded$
    type=counter
    'help=The total number of client requests completed in an HTTP location zone␣
↪without sending a response.';


'angie_http_location_zones_responses{zone="$1",code="$2"}' $p8s_value
    path=~^/http/location_zones/([^/]+)/responses/([^/]+)$
    type=counter
    'help=The number of responses with a specific status in an HTTP location zone.';


'angie_http_location_zones_data_received{zone="$1"}' $p8s_value
    path=~^/http/location_zones/([^/]+)/data/received$
    type=counter
    'help=The total number of bytes received from clients in an HTTP location zone.';

'angie_http_location_zones_data_sent{zone="$1"}' $p8s_value
    path=~^/http/location_zones/([^/]+)/data/sent$
    type=counter
    'help=The total number of bytes sent to clients in an HTTP location zone.';


'angie_http_upstreams_peers_backup{upstream="$1",peer="$2"}' $p8st_all_ups_backup
    path=~^/http/upstreams/([^/]+)/peers/([^/]+)/backup$
    type=gauge
    'help=The HTTP upstream peer backup group level.';


'angie_http_upstreams_peers_state{upstream="$1",peer="$2"}' $p8st_all_ups_state
    path=~^/http/upstreams/([^/]+)/peers/([^/]+)/state$
    type=gauge
    'help=The current state of an upstream peer in "HTTP": 1 - up, 2 - down, 3 -␣
↪unavailable, 4 - recovering, 5 - unhealthy, 6 - checking, or 7 - draining.';


'angie_http_upstreams_peers_selected_current{upstream="$1",peer="$2"}' $p8s_value
    path=~^/http/upstreams/([^/]+)/peers/([^/]+)/selected/current$
    type=gauge
    'help=The number of requests currently being processed by an upstream peer in
↪"HTTP".';

'angie_http_upstreams_peers_selected_total{upstream="$1",peer="$2"}' $p8s_value
    path=~^/http/upstreams/([^/]+)/peers/([^/]+)/selected/total$
    type=counter
    'help=The total number of attempts to use an upstream peer in "HTTP".';


'angie_http_upstreams_peers_responses{upstream="$1",peer="$2",code="$3"}' $p8s_value
```

```
    path=~^/http/upstreams/([^/]+)/peers/([^/]+)/responses/([^/]+)$
    type=counter
    'help=The number of responses with a specific status received from an upstream␣
↪peer in "HTTP".';


'angie_http_upstreams_peers_data_sent{upstream="$1",peer="$2"}' $p8s_value
    path=~^/http/upstreams/([^/]+)/peers/([^/]+)/data/sent$
    type=counter
    'help=The total number of bytes sent to an upstream peer in "HTTP".';

'angie_http_upstreams_peers_data_received{upstream="$1",peer="$2"}' $p8s_value
    path=~^/http/upstreams/([^/]+)/peers/([^/]+)/data/received$
    type=counter
    'help=The total number of bytes received from an upstream peer in "HTTP".';


'angie_http_upstreams_peers_health_fails{upstream="$1",peer="$2"}' $p8s_value
    path=~^/http/upstreams/([^/]+)/peers/([^/]+)/health/fails$
    type=counter
    'help=The total number of unsuccessful attempts to communicate with an upstream␣
↪peer in "HTTP".';

'angie_http_upstreams_peers_health_unavailable{upstream="$1",peer="$2"}' $p8s_value
    path=~^/http/upstreams/([^/]+)/peers/([^/]+)/health/unavailable$
    type=counter
    'help=The number of times when an upstream peer in "HTTP" became "unavailable"␣
↪due to reaching the max_fails limit.';

'angie_http_upstreams_peers_health_downtime{upstream="$1",peer="$2"}' $p8s_value
    path=~^/http/upstreams/([^/]+)/peers/([^/]+)/health/downtime$
    type=counter
    'help=The total time (in milliseconds) that an upstream peer in "HTTP" was
↪"unavailable".';

'angie_http_upstreams_peers_health_header_time{upstream="$1",peer="$2"}' $p8s_value
    path=~^/http/upstreams/([^/]+)/peers/([^/]+)/health/header_time$
    type=gauge
    'help=Average time (in milliseconds) to receive the response headers from an␣
↪upstream peer in "HTTP".';

'angie_http_upstreams_peers_health_response_time{upstream="$1",peer="$2"}' $p8s_value
    path=~^/http/upstreams/([^/]+)/peers/([^/]+)/health/response_time$
    type=gauge
    'help=Average time (in milliseconds) to receive the complete response from an␣
↪upstream peer in "HTTP".';

'angie_http_upstreams_peers_health_probes_count{upstream="$1",peer="$2"}' $p8s_value
    path=~^/http/upstreams/([^/]+)/peers/([^/]+)/health/probes/count$
    type=counter
    'help=The total number of probes for this peer.';

'angie_http_upstreams_peers_health_probes_fails{upstream="$1",peer="$2"}' $p8s_value
    path=~^/http/upstreams/([^/]+)/peers/([^/]+)/health/probes/fails$
    type=counter
    'help=The total number of failed probes for this peer.';
```

```
'angie_http_upstreams_keepalive{upstream="$1"}' $p8s_value
    path=~^/http/upstreams/([^/]+)/keepalive$
    type=gauge
    'help=The number of currently cached keepalive connections for an HTTP upstream.';


'angie_http_upstreams_backup_switch_active{upstream="$1"}' $p8s_value
    path=~^/http/upstreams/([^/]+)/backup_switch/active$
    type=gauge
    'help=The currently active HTTP upstream servers backup group level.';


'angie_http_upstreams_queue_queued{upstream="$1"}' $p8s_value
    path=~^/http/upstreams/([^/]+)/queue/queued$
    type=counter
    'help=The total number of queued requests for an HTTP upstream.';

'angie_http_upstreams_queue_waiting{upstream="$1"}' $p8s_value
    path=~^/http/upstreams/([^/]+)/queue/waiting$
    type=gauge
    'help=The number of requests currently waiting in an HTTP upstream queue.';

'angie_http_upstreams_queue_dropped{upstream="$1"}' $p8s_value
    path=~^/http/upstreams/([^/]+)/queue/dropped$
    type=counter
    'help=The total number of requests dropped from an HTTP upstream queue because␣
↪the client had prematurely closed the connection.';

'angie_http_upstreams_queue_timedout{upstream="$1"}' $p8s_value
    path=~^/http/upstreams/([^/]+)/queue/timedout$
    type=counter
    'help=The total number of requests timed out from an HTTP upstream queue.';

'angie_http_upstreams_queue_overflows{upstream="$1"}' $p8s_value
    path=~^/http/upstreams/([^/]+)/queue/overflows$
    type=counter
    'help=The total number of requests rejected by an HTTP upstream queue because the␣
↪size limit had been reached.';


'angie_http_caches_responses{zone="$1",status="$2"}' $p8s_value
    path=~^/http/caches/([^/]+)/([^/]+)/responses$
    type=counter
    'help=The total number of responses processed in an HTTP cache zone with a␣
↪specific cache status.';

'angie_http_caches_bytes{zone="$1",status="$2"}' $p8s_value
    path=~^/http/caches/([^/]+)/([^/]+)/bytes$
    type=counter
    'help=The total number of bytes processed in an HTTP cache zone with a specific␣
↪cache status.';

'angie_http_caches_responses_written{zone="$1",status="$2"}' $p8s_value
    path=~^/http/caches/([^/]+)/([^/]+)/responses_written$
    type=counter
    'help=The total number of responses written to an HTTP cache zone with a specific␣
```

```
↪cache status.';

'angie_http_caches_bytes_written{zone="$1",status="$2"}' $p8s_value
    path=~^/http/caches/([^/]+)/([^/]+)/bytes_written$
    type=counter
    'help=The total number of bytes written to an HTTP cache zone with a specific␣
↪cache status.';


'angie_http_caches_size{zone="$1"}' $p8s_value
    path=~^/http/caches/([^/]+)/size$
    type=gauge
    'help=The current size (in bytes) of cached responses in an HTTP cache zone.';


'angie_http_caches_shards_size{zone="$1",path="$2"}' $p8s_value
    path=~^/http/caches/([^/]+)/shards/([^/]+)/size$
    type=gauge
    'help=The current size (in bytes) of cached responses in a shard path of an HTTP␣
↪cache zone.';


'angie_http_limit_conns{zone="$1",status="$2"}' $p8s_value
    path=~^/http/limit_conns/([^/]+)/([^/]+)$
    type=counter
    'help=The number of requests processed by an HTTP limit_conn zone with a specific␣
↪result.';

'angie_http_limit_reqs{zone="$1",status="$2"}' $p8s_value
    path=~^/http/limit_reqs/([^/]+)/([^/]+)$
    type=counter
    'help=The number of requests processed by an HTTP limit_reqs zone with a specific␣
↪result.';


'angie_stream_server_zones_ssl_handshaked{zone="$1"}' $p8s_value
    path=~^/stream/server_zones/([^/]+)/ssl/handshaked$
    type=counter
    'help=The total number of successful SSL handshakes in a stream server zone.';

'angie_stream_server_zones_ssl_reuses{zone="$1"}' $p8s_value
    path=~^/stream/server_zones/([^/]+)/ssl/reuses$
    type=counter
    'help=The total number of session reuses during SSL handshakes in a stream server␣
↪zone.';

'angie_stream_server_zones_ssl_timedout{zone="$1"}' $p8s_value
    path=~^/stream/server_zones/([^/]+)/ssl/timedout$
    type=counter
    'help=The total number of timed-out SSL handshakes in a stream server zone.';

'angie_stream_server_zones_ssl_failed{zone="$1"}' $p8s_value
    path=~^/stream/server_zones/([^/]+)/ssl/failed$
    type=counter
    'help=The total number of failed SSL handshakes in a stream server zone.';
```

```
'angie_stream_server_zones_connections_total{zone="$1"}' $p8s_value
    path=~^/stream/server_zones/([^/]+)/connections/total$
    type=counter
    'help=The total number of client connections received in a stream server zone.';

'angie_stream_server_zones_connections_processing{zone="$1"}' $p8s_value
    path=~^/stream/server_zones/([^/]+)/connections/processing$
    type=gauge
    'help=The number of client connections currently being processed in a stream␣
↪server zone.';

'angie_stream_server_zones_connections_discarded{zone="$1"}' $p8s_value
    path=~^/stream/server_zones/([^/]+)/connections/discarded$
    type=counter
    'help=The total number of client connections completed in a stream server zone␣
↪without establishing a session.';

'angie_stream_server_zones_connections_passed{zone="$1"}' $p8s_value
    path=~^/stream/server_zones/([^/]+)/connections/passed$
    type=counter
    'help=The total number of client connections in a stream server zone passed for␣
↪handling to a different listening socket.';


'angie_stream_server_zones_sessions{zone="$1",status="$2"}' $p8s_value
    path=~^/stream/server_zones/([^/]+)/sessions/([^/]+)$
    type=counter
    'help=The number of sessions finished with a specific status in a stream server␣
↪zone.';


'angie_stream_server_zones_data_received{zone="$1"}' $p8s_value
    path=~^/stream/server_zones/([^/]+)/data/received$
    type=counter
    'help=The total number of bytes received from clients in a stream server zone.';

'angie_stream_server_zones_data_sent{zone="$1"}' $p8s_value
    path=~^/stream/server_zones/([^/]+)/data/sent$
    type=counter
    'help=The total number of bytes sent to clients in a stream server zone.';


'angie_stream_upstreams_peers_backup{upstream="$1",peer="$2"}' $p8st_all_ups_backup
    path=~^/stream/upstreams/([^/]+)/peers/([^/]+)/backup$
    type=gauge
    'help=The "stream" upstream peer backup group level.';


'angie_stream_upstreams_peers_state{upstream="$1",peer="$2"}' $p8st_all_ups_state
    path=~^/stream/upstreams/([^/]+)/peers/([^/]+)/state$
    type=gauge
    'help=The current state of an upstream peer in "stream": 1 - up, 2 - down, 3 -␣
↪unavailable, 4 - recovering, 5 - unhealthy, 6 - checking, or 7 - draining.';


'angie_stream_upstreams_peers_selected_current{upstream="$1",peer="$2"}' $p8s_value
    path=~^/stream/upstreams/([^/]+)/peers/([^/]+)/selected/current$
```

```
    type=gauge
    'help=The number of sessions currently being processed by an upstream peer in
↪"stream".';

'angie_stream_upstreams_peers_selected_total{upstream="$1",peer="$2"}' $p8s_value
    path=~^/stream/upstreams/([^/]+)/peers/([^/]+)/selected/total$
    type=counter
    'help=The total number of attempts to use an upstream peer in "stream".';


'angie_stream_upstreams_peers_data_sent{upstream="$1",peer="$2"}' $p8s_value
    path=~^/stream/upstreams/([^/]+)/peers/([^/]+)/data/sent$
    type=counter
    'help=The total number of bytes sent to an upstream peer in "stream".';

'angie_stream_upstreams_peers_data_received{upstream="$1",peer="$2"}' $p8s_value
    path=~^/stream/upstreams/([^/]+)/peers/([^/]+)/data/received$
    type=counter
    'help=The total number of bytes received from an upstream peer in "stream".';

'angie_stream_upstreams_peers_data_pkt_sent{upstream="$1",peer="$2"}' $p8s_value
    path=~^/stream/upstreams/([^/]+)/peers/([^/]+)/data/pkt_sent$
    type=counter
    'help=The total number of packets sent to an upstream peer in "stream".';

'angie_stream_upstreams_peers_data_pkt_received{upstream="$1",peer="$2"}' $p8s_value
    path=~^/stream/upstreams/([^/]+)/peers/([^/]+)/data/pkt_received$
    type=counter
    'help=The total number of packets received from an upstream peer in "stream".';


'angie_stream_upstreams_peers_health_fails{upstream="$1",peer="$2"}' $p8s_value
    path=~^/stream/upstreams/([^/]+)/peers/([^/]+)/health/fails$
    type=counter
    'help=The total number of unsuccessful attempts to communicate with an upstream␣
↪peer in "stream".';

'angie_stream_upstreams_peers_health_unavailable{upstream="$1",peer="$2"}' $p8s_value
    path=~^/stream/upstreams/([^/]+)/peers/([^/]+)/health/unavailable$
    type=counter
    'help=The number of times when an upstream peer in "stream" became "unavailable"␣
↪due to reaching the max_fails limit.';

'angie_stream_upstreams_peers_health_downtime{upstream="$1",peer="$2"}' $p8s_value
    path=~^/stream/upstreams/([^/]+)/peers/([^/]+)/health/downtime$
    type=counter
    'help=The total time (in milliseconds) that an upstream peer in "stream" was
↪"unavailable".';

'angie_stream_upstreams_peers_health_connect_time{upstream="$1",peer="$2"}' $p8s_value
    path=~^/stream/upstreams/([^/]+)/peers/([^/]+)/health/connect_time$
    type=gauge
    'help=Average time (in milliseconds) to connect to an upstream peer in "stream".';

'angie_stream_upstreams_peers_health_first_byte_time{upstream="$1",peer="$2"}' $p8s_
↪value
    path=~^/stream/upstreams/([^/]+)/peers/([^/]+)/health/first_byte_time$
```

```
    type=gauge
    'help=Average time (in milliseconds) to receive the first byte from an upstream␣
↪peer in "stream".';

'angie_stream_upstreams_peers_health_last_byte_time{upstream="$1",peer="$2"}' $p8s_
↪value
    path=~^/stream/upstreams/([^/]+)/peers/([^/]+)/health/last_byte_time$
    type=gauge
    'help=Average time (in milliseconds) of the whole communication session with an␣
↪upstream peer in "stream".';


'angie_stream_upstreams_peers_health_probes_count{upstream="$1",peer="$2"}' $p8s_value
    path=~^/stream/upstreams/([^/]+)/peers/([^/]+)/health/probes/count$
    type=counter
    'help=The total number of probes for this peer.';

'angie_stream_upstreams_peers_health_probes_fails{upstream="$1",peer="$2"}' $p8s_value
    path=~^/stream/upstreams/([^/]+)/peers/([^/]+)/health/probes/fails$
    type=counter
    'help=The total number of failed probes for this peer.';
}

map $p8s_value $p8st_all_ups_state {
    volatile;
    "up"           1;
    "down"         2;
    "unavailable"  3;
    "recovering"   4;
    "unhealthy"    5;
    "checking"     6;
    "draining"     7;
    "busy"         8;
    default        0;
}

map $p8s_value $p8st_all_ups_backup {
    volatile;
    "false"        0;
    "true"         1;
    default        $p8s_value;
}
```

Usage:

```
http {

    include prometheus_all.conf;

    #...

    server {

        listen 80;

        location =/p8s {
            prometheus all;
        }
```

```
        # ...

    }
}
```

```
$ curl localhost/p8s

    # Angie Prometheus template "all"
    ...
```

Directives

prometheus

Syntax	`prometheus` *template*;
Default	—
Context	location

Sets the handler template for the `location` context; the template should be defined by a *prometheus_template* directive. Upon a request, this `location` evaluates and returns the template metrics in the Prometheus format.

```
location =/p8s {
    prometheus custom;
}
```

```
$ curl localhost/p8s

    # Angie Prometheus template "custom"
    ...
```

prometheus_template

Syntax	`prometheus_template` *template* { ... }
Default	—
Context	http

Defines a named template of metrics to be collected and exported by Angie; the template should be used with a *prometheus* directive.

Note

Angie also has a predefined *all* template that includes a number of common metrics.

The template can contain any number of metrics definitions, each structured as follows: <*metric*> <*variable*> [`path=`<*match*>] [`type=`<*type*>] [`help=`<*help*>];

`metric`	Sets the name of the metric to be added to the response in the Prometheus format. Can contain an optional label definition (`...`), for example: `http_requests_total{method="$1",code="$2"}` Label values can use Angie variables; if *match* is a regex, you can also use its capture groups with labels. Such variables and groups are evaluated when the metric's *variable* itself is evaluated.
`variable`	Sets the name of the variable to be evaluated and added to the response as the metric value. If there's no such variable or the resulting value is empty (`""`), no metric is added.

The metric's value is evaluated from the *variable*; upon a successful evaluation, the metric is added to the response, for example:

```
'angie_time{version="$angie_version"}' $msec;
```

```
$ curl localhost/p8s

    angie_time{version="1.9.1"} 1695119820.562
```

`path=match`	Is matched against all end-to-end metric paths in the */status* section of Angie API, allowing to add several metric instances to the response at once.

During matching, the paths include the starting slash but not the ending one, for example: `/angie/generation`; matching is case-insensitive. There are two matching modes:

`path=exact_match`	Matches strings character by character.
`path=~regex_matc`	PCRE-based match; can define capture groups to be used with the *metric* name labels.

If *match* matches a path, the value of the Angie metric at this path is stored in the *$p8s_value* variable, which can be used as *variable* when `path=` is set.

When regex match is used, multiple matches can occur; in this case, a metric is added to the response for *each* matching path. With capture groups, this enables series of metrics that share one name but have different labels, for example:

```
'angie_slabs_slots_free{zone="$1",size="$2"}' $p8s_value
    path=~^/slabs/([^/]+)/slots/([^/]+)/free$;
```

This adds a metric for each zone and size combination the configuration contains:

```
angie_slabs_slots_free{zone="one",size="8"} 502
angie_slabs_slots_free{zone="one",size="16"} 249
angie_slabs_slots_free{zone="one",size="32"} 122
angie_slabs_slots_free{zone="one",size="128"} 22
angie_slabs_slots_free{zone="one",size="512"} 4
angie_slabs_slots_free{zone="two",size="8"} 311
...
```

If there are no matches (regardless of the mode), no metric is added.

Note

The `path=` option is available only when Angie is built with the *API* module.

`type=type`, `help=help`	Set the metric's type and help string, respectively, using the Prometheus format; both are added to the response with the metric itself without any transformation or validation.

Built-in Variables

The `http_prometheus` module has a built-in variable that receives its value when the paths from the */status* section of Angie API are matched against the *match* option of a metric defined by the *prometheus_ template* directive.

`$p8s_value`

If the *match* of a metric defined by *prometheus_ template* matches a path, the value of the Angie metric at this path is stored in the `$p8s_value` variable. It's intended to be used as the *variable* in `path=`-based metric definitions.

The values of Angie metrics stored in `$p8s_value` may deviate from the requirements of the Prometheus format. In this case, the *map* directive can help transform strings into numbers:

```
map $p8s_value $ups_state_n {
    up           0;
    unavailable  1;
    down         2;
    default      3;
}

prometheus_template main {
    'angie_http_upstreams_state{upstream="$1",peer="$2"}' $ups_state_n
        path=~^/http/upstreams/([^/]+)/peers/([^/]+)/state$;
}
```

If the Angie metric is Boolean, namely, `true` or `false`, the variable is set to `"1"` or `"0"`, respectively; if the metric is `null`, the variable is set to `"(null)"`. For date values, the integer epoch format is used.

Proxy

Allows passing requests to another (proxied) server.

Configuration Example

```
location / {
    proxy_pass       http://localhost:8000;
    proxy_set_header Host      $host;
    proxy_set_header X-Real-IP $remote_addr;
}
```

Directives

proxy_bind

Syntax	`proxy_bind` *address* [`transparent`] | `off`;
Default	—
Context	http, server, location

Makes outgoing connections to a proxied server originate from the specified local IP address with an optional port. Parameter value can contain variables. The special value `off` cancels the effect of the

proxy_bind directive inherited from the previous configuration level, which allows the system to auto-assign the local IP address and port.

The `transparent` parameter allows outgoing connections to a proxied server originate from a non-local IP address, for example, from a real IP address of a client:

```
proxy_bind $remote_addr transparent;
```

In order for this parameter to work, it is usually necessary to run Angie worker processes with the *superuser* privileges. On Linux it is not required as if the `transparent` parameter is specified, worker processes inherit the *CAP_NET_RAW* capability from the master process.

Important

It is necessary to configure kernel routing table to intercept network traffic from the proxied server.

proxy_buffer_size

Syntax	`proxy_buffer_size` *size*;
Default	`proxy_buffer_size 4k|8k;`
Context	http, server, location

Sets the size of the buffer used for reading the first part of the response received from the proxied server. This part usually contains a small response header. By default, the buffer size is equal to one memory page. This is either 4K or 8K, depending on a platform. It can be made smaller, however.

proxy_buffering

Syntax	`proxy_buffering on | off;`
Default	`proxy_buffering on;`
Context	http, server, location

Enables or disables buffering of responses from the proxied server.

`on`	Angie receives a response from the proxied server as soon as possible, saving it into the buffers set by the *proxy_buffer_size* and *proxy_buffers* directives. If the whole response does not fit into memory, a part of it can be saved to a *temporary file* on the disk. Writing to temporary files is controlled by the *proxy_max_temp_file_size* and *proxy_temp_file_write_size* directives.
`off`	The response is passed to a client synchronously, immediately as it is received. Angie will not try to read the whole response from the proxied server. The maximum size of the data that Angie can receive from the server at a time is set by the *proxy_buffer_size* directive.

Buffering can also be enabled or disabled by passing "yes" or "no" in the "X-Accel-Buffering" response header field. This capability can be disabled using the *proxy_ignore_headers* directive.

proxy_buffers

Syntax	`proxy_buffers` *number size*;
Default	`proxy_buffers 8 4k | 8k;`
Context	http, server, location

Sets the number and size of the buffers used for reading a response from the proxied server, for a single connection.

By default, the buffer size is equal to one memory page. This is either 4K or 8K, depending on a platform.

proxy_busy_buffers_size

Syntax	`proxy_busy_buffers_size` *size*;
Default	`proxy_busy_buffers_size 8k | 16k;`
Context	http, server, location

When *buffering* of responses from the proxied server is enabled, limits the total size of buffers that can be busy sending a response to the client while the response is not yet fully read. In the meantime, the rest of the buffers can be used for reading the response and, if needed, buffering part of the response to a temporary file.

By default, size is limited by the size of two buffers set by the *proxy_buffer_size* and *proxy_buffers* directives.

proxy_cache

Syntax	`proxy_cache` *zone* | `off` [`path=`*path*];
Default	`proxy_cache off;`
Context	http, server, location

Defines a shared memory zone for caching. A zone can be used in the configuration multiple times. The parameter's value allows variables.

`off`	disables caching inherited from the previous configuration level.

Added in version 1.2.0: PRO

In Angie PRO, you can specify multiple *proxy_cache_path* directives that share the same `keys_zone` value to implement *cache sharding*. If you do, set the `path` parameter of the *proxy_cache* directive that references this `keys_zone`:

`path=`*path*	The value is determined when the backend's response is *cached*, which implies that variables are involved, including those that store some information from the response. If the response is obtained from the cache, *path* isn't reevaluated; thus, a response from the cache will preserve its original *path* until it's deleted from the cache.

This allows choosing between cache paths by applying *map* directives or scripts to responses from the backend. A `Content-Type` example:

```
proxy_cache_path /cache/one keys_zone=zone:10m;
proxy_cache_path /cache/two keys_zone=zone;

map $upstream_http_content_type $cache {
   ~^text/  one;
   default  two;
}

server {
   ...
```

```
    location / {
        proxy_pass http://backend;
        proxy_cache zone path=/cache/$cache;
    }
}
```

This adds two cache paths and a variable mapping to choose between them. If `Content-Type` starts with `text/`, the first path is used; otherwise, the second.

proxy_cache_background_update

Syntax	`proxy_cache_background_update on | off;`
Default	`proxy_cache_background_update off;`
Context	http, server, location

Allows starting a background subrequest to update an expired cache item, while a stale cached response is returned to the client.

Attention

Note that it is necessary to *allow* the usage of a stale cached response when it is being updated.

proxy_cache_bypass

Syntax	`proxy_cache_bypass ...;`
Default	—
Context	http, server, location

Defines conditions under which the response will not be taken from a cache. If at least one value of the string parameters is not empty and is not equal to "0" then the response will not be taken from the cache:

```
proxy_cache_bypass $cookie_nocache $arg_nocache$arg_comment;
proxy_cache_bypass $http_pragma    $http_authorization;
```

Can be used along with the *proxy_no_cache* directive.

proxy_cache_convert_head

Syntax	`proxy_cache_convert_head on | off;`
Default	`proxy_cache_convert_head on;`
Context	http, server, location

Enables or disables the conversion of the "HEAD" method to "GET" for caching. When the conversion is disabled, the *cache key* should be configured to include the *$request_method*.

proxy_cache_key

Syntax	`proxy_cache_key` *string*;
Default	`proxy_cache_key $scheme$proxy_host$request_uri;`
Context	http, server, location

Defines a key for caching, for example

```
proxy_cache_key "$host$request_uri $cookie_user";
```

By default, the directive's value is close to the string

```
proxy_cache_key $scheme$proxy_host$uri$is_args$args;
```

proxy_cache_lock

Syntax	`proxy_cache_lock on	off;`
Default	`proxy_cache_lock off;`
Context	http, server, location

When enabled, only one request at a time will be allowed to populate a new cache element identified according to the *proxy_cache_key* directive by passing a request to a proxied server. Other requests of the same cache element will either wait for a response to appear in the cache or the cache lock for this element to be released, up to the time set by the *proxy_cache_lock_timeout* directive.

proxy_cache_lock_age

Syntax	`proxy_cache_lock_age` *time*;
Default	`proxy_cache_lock_age 5s;`
Context	http, server, location

If the last request passed to the proxied server for populating a new cache element has not completed for the specified time, one more request may be passed to the proxied server.

proxy_cache_lock_timeout

Syntax	`proxy_cache_lock_timeout` *time*;
Default	`proxy_cache_lock_timeout 5s;`
Context	http, server, location

Sets a timeout for *proxy_cache_lock*. When the time expires, the request will be passed to the proxied server, however, the response will not be cached.

proxy_cache_max_range_offset

Syntax	`proxy_cache_max_range_offset` *number*;
Default	—
Context	http, server, location

Sets an offset in bytes for byte-range requests. If the range is beyond the offset, the range request will be passed to the proxied server and the response will not be cached.

proxy_cache_methods

Syntax	`proxy_cache_methods GET	HEAD	POST ...;`
Default	`proxy_cache_methods GET HEAD;`
Context	http, server, location

If the client request method is listed in this directive then the response will be cached. "GET" and "HEAD" methods are always added to the list, though it is recommended to specify them explicitly. See also the *proxy_no_cache* directive.

proxy_cache_min_uses

Syntax	`proxy_cache_min_uses` *number*;
Default	`proxy_cache_min_uses 1;`
Context	http, server, location

Sets the number of requests after which the response will be cached.

proxy_cache_path

Syntax	`proxy_cache_path` *path* [`levels=`*levels*] [`use_temp_path=on` | `off`] `keys_zone=`*name:size*[`:file=`*file*] [`inactive=`*time*] [`max_size=`*size*] [`min_free=`*size*] [`manager_files=`*number*] [`manager_sleep=`*time*] [`manager_threshold=`*time*] [`loader_files=`*number*] [`loader_sleep=`*time*] [`loader_threshold=`*time*];
Default	—
Context	http

Sets the *path* and other parameters of a cache. Cache data are stored in files. The file name in a cache is a result of applying the MD5 function to the *cache key*.

`levels`	defines hierarchy levels of a cache: from 1 to 3, each level accepts values 1 or 2.

For example, in the following configuration:

```
proxy_cache_path /data/angie/cache levels=1:2 keys_zone=one:10m;
```

file names in a cache will look like this:

```
/data/angie/cache/c/29/b7f54b2df7773722d382f4809d65029c
```

A cached response is first written to a temporary file, and then the file is renamed. Temporary files and the cache can be put on different file systems. However, be aware that in this case a file is copied across two file systems instead of the cheap renaming operation. It is thus recommended that for any given location both cache and a directory holding temporary files are put on the same file system.

`use_temp_path=on`	Sets the directory for temporary files
`on`	If this parameter is omitted or set to the value `on`, the directory set by the *proxy_temp_path* directive for the given *location* will be used.
`off`	Temporary files will be put directly in the cache directory.
`keys_zone`	Configures the name and size for a shared memory zone to store all active keys and information about data. One megabyte zone can store about 8,000 keys. When the optional *file* parameter is used with `keys_zone`, Angie flushes the contents of this zone to the disk on master process exit and attempts to restore it at the same memory address at next *startup* or after a *binary upgrade*; to achieve more robust persistence and improve cache loading time. If the zone cannot be restored due to a change in size, binary version incompatibility, or other reasons, Angie will log an alert (`failed to restore zone at address`) and will not use the zone restore mechanism. Instead, the incompatible file will be renamed as `.old`; you can either delete it, or restore its name and revert Angie to the configuration and version where it was created in the first place. **Attention** Ensure that the *file* path is valid and has the correct permissions for Angie to use it and prevent unauthorized access at the same time; relative paths are prefix-based.
`inactive`	Cached data that are not accessed during the time specified by this parameter get removed from the cache regardless of their freshness. By default, it is set to 10 minutes.

Note

Added in version 1.2.0: PRO

In Angie PRO, multiple *proxy_cache_path* directives that share the same `keys_zone` value are allowed. Only the first such directive may set the shared memory zone size. The choice between such directives is made by the `path` parameter of the relevant *proxy_cache* directive.

A special **cache manager** process monitors the maximum cache size and the minimum amount of free space on the file system with cache and when the size is exceeded or there is not enough free space, it removes the least recently used data. The data is removed in iterations.

`max_size`	maximum cache size
`min_free`	minimum amount of free space on the file system with cache
`manager_files`	limits the number of items to be deleted during one iteration By default, `100`
`manager_threshol`	limits the duration of one iteration By default, `200` milliseconds
`manager_sleep`	configures a pause between interactions By default, `50` milliseconds

A minute after Angie starts, the special **cache loader** process is activated. It scans the file system for previously cached data and loads that information into the cache zone. This process is carried out in iterations; each iteration processes a limited number of items set by `loader_files`, ensures it does not exceed the `loader_threshold`, then pauses for a short interval set by `loader_sleep` before proceeding to the next batch. These iterations continue until the loader has processed all existing cache entries on disk:

`loader_files`	limits the number of items to load during one iteration By default, `100`
`loader_threshold`	limits the duration of one iteration By default, `200` milliseconds
`loader_sleep`	configures a pause between iterations By default, `50` milliseconds

Note

Setting the *file* path for the `keys_zone` parameter doesn't interfere with the cache loader behavior.

proxy_cache_revalidate

Syntax	`proxy_cache_revalidate on | off;`
Default	`proxy_cache_revalidate off;`
Context	http, server, location

Enables revalidation of expired cache items using conditional requests with the "If-Modified-Since" and "If-None-Match" header fields.

proxy_cache_use_stale

Syntax	`proxy_cache_use_stale error | timeout | invalid_header | updating | http_500 | http_502 | http_503 | http_504 | http_403 | http_404 | http_429 | off ...;`
Default	`proxy_cache_use_stale off;`
Context	http, server, location

Determines in which cases a stale cached response can be used during communication with the proxied server. The directive's parameters match the parameters of the *proxy_next_upstream* directive.

`error`	permits using a stale cached response if a proxied server to process a request cannot be selected.
`updating`	additional parameter, permits using a stale cached response if it is currently being updated. This allows minimizing the number of accesses to proxied servers when updating cached data.

Using a stale cached response can also be enabled directly in the response header for a specified number of seconds after the response became stale:

- The stale-while-revalidate extension of the "Cache-Control" header field permits using a stale cached response if it is currently being updated.
- The stale-if-error extension of the "Cache-Control" header field permits using a stale cached response in case of an error.

Note

This has lower priority than using the directive parameters.

To minimize the number of accesses to proxied servers when populating a new cache element, the *proxy_cache_lock* directive can be used.

proxy_cache_valid

Syntax	`proxy_cache_valid` [*code* ...] *time*;
Default	—
Context	http, server, location

Sets caching time for different response codes. For example, the following directives

```
proxy_cache_valid 200 302 10m;
proxy_cache_valid 404      1m;
```

set 10 minutes of caching for responses with codes 200 and 302 and 1 minute for responses with code 404.

If only caching time is specified

```
proxy_cache_valid 5m;
```

then only 200, 301, and 302 responses are cached.

In addition, the `any` parameter can be specified to cache any responses:

```
proxy_cache_valid 200 302 10m;
proxy_cache_valid 301      1h;
proxy_cache_valid any      1m;
```

Note

Parameters of caching can also be set directly in the response header. This has higher priority than setting of caching time using the directive

- The "X-Accel-Expires" header field sets caching time of a response in seconds. The zero value disables caching for a response. If the value starts with the @ prefix, it sets an absolute time in seconds since Epoch, up to which the response may be cached.
- If the header does not include the "X-Accel-Expires" field, parameters of caching may be set in the header fields "Expires" or "Cache-Control".
- If the header includes the "Set-Cookie" field, such a response will not be cached.
- If the header includes the "Vary" field with the special value "*", such a response will not be cached. If the header includes the "Vary" field with another value, such a response will be cached taking into account the corresponding request header fields.

Processing of one or more of these response header fields can be disabled using the *proxy_ignore_headers* directive.

proxy_connect_timeout

Syntax	`proxy_connect_timeout` *time*;
Default	`proxy_connect_timeout 60s;`
Context	http, server, location

Defines a timeout for establishing a connection with a proxied server. It should be noted that this timeout cannot usually exceed 75 seconds.

proxy_connection_drop

Syntax	`proxy_connection_drop` *time* | `on` | `off`;
Default	`proxy_connection_drop off;`
Context	http, server, location

Enables termination of all connections to the proxied server after it has been removed from the group or marked as permanently unavailable by a *reresolve* process or the *API command* `DELETE`.

A connection is terminated when the next read or write event is processed for either the client or the proxied server.

Setting *time* enables a connection termination *timeout*; with `on` set, connections are dropped immediately.

proxy_cookie_domain

Syntax	`proxy_cookie_domain off;` `proxy_cookie_domain` *domain replacement*;
Default	`proxy_cookie_domain off;`
Context	http, server, location

Sets a text that should be changed in the domain attribute of the "Set-Cookie" header fields of a proxied server response. Suppose a proxied server returned the "Set-Cookie" header field with the attribute "domain=localhost". The directive

```
proxy_cookie_domain localhost example.org;
```

will rewrite this attribute to "domain=example.org".

A dot at the beginning of the *domain* and *replacement* strings and the domain attribute is ignored. Matching is case-insensitive.

The *domain* and *replacement* strings can contain variables:

```
proxy_cookie_domain www.$host $host;
```

The directive can also be specified using regular expressions. In this case, *domain* should start with a "~" symbol. A regular expression can contain named and positional captures, and *replacement* can reference them:

```
proxy_cookie_domain ~\.(?P<sl_domain>[-0-9a-z]+\.[a-z]+)$ $sl_domain;
```

Multiple *proxy_ cookie_ domain* directives may be specified at the same level:

```
proxy_cookie_domain localhost example.org;
proxy_cookie_domain ~\.([a-z]+\.[a-z]+)$ $1;
```

If several directives can be applied to the cookie, the first matching directive will be chosen.

The `off` parameter cancels the effect of the *proxy_ cookie_ domain* directives inherited from the previous configuration level.

proxy_cookie_flags

Syntax	`proxy_cookie_flags off` | *cookie* [*flag* ...];
Default	`proxy_cookie_flags off;`
Context	http, server, location

Sets one or more flags for the cookie. The cookie can contain text, variables, and their combinations. The flag can contain text, variables, and their combinations.

The `secure`, `httponly`, `samesite=strict`, `samesite=lax`, `samesite=none` parameters add the corresponding flags.

The `nosecure`, `nohttponly`, `nosamesite` parameters remove the corresponding flags.

The cookie can also be specified using regular expressions. In this case, cookie should start with a "`~`" symbol.

Several `proxy_cookie_flags` directives can be specified on the same configuration level:

```
proxy_cookie_flags one httponly;
proxy_cookie_flags ~ nosecure samesite=strict;
```

If several directives can be applied to the cookie, the first matching directive will be chosen. In the example, the *httponly* flag is added to the cookie *one*, for all other cookies the *samesite=strict* flag is added and the *secure* flag is deleted.

The `off` parameter cancels the effect of the *proxy_cookie_flags* directives inherited from the previous configuration level.

proxy_cookie_path

Syntax	`proxy_cookie_path off;` `proxy_cookie_path` *path replacement*;
Default	`proxy_cookie_path off;`
Context	http, server, location

Sets a text that should be changed in the path attribute of the `Set-Cookie` header fields of a proxied server response. Suppose a proxied server returned the "Set-Cookie" header field with the attribute "path=/two/some/uri/". The directive

```
proxy_cookie_path /two/ /;
```

will rewrite this attribute to "path=/some/uri/".

The *path* and *replacement* strings can contain variables:

```
proxy_cookie_path $uri /some$uri;
```

The directive can also be specified using regular expressions. In this case, *path* should either start with a "`~`" symbol for a case-sensitive matching, or with the "`~*`" symbols for case-insensitive matching. The regular expression can contain named and positional captures, and *replacement* can reference them:

```
proxy_cookie_path ~*^/user/([^/]+) /u/$1;
```

Several *proxy_cookie_path* directives can be specified on the same level:

```
proxy_cookie_path /one/ /;
proxy_cookie_path / /two/;
```

If several directives can be applied to the cookie, the first matching directive will be chosen.

The `off` parameter cancels the effect of the *proxy_cookie_path* directives inherited from the previous configuration level.

proxy_force_ranges

Syntax	`proxy_force_ranges off;`
Default	`proxy_force_ranges off;`
Context	http, server, location

Enables byte-range support for both cached and uncached responses from the proxied server regardless of the "Accept-Ranges" field in these responses.

proxy_headers_hash_bucket_size

Syntax	`proxy_headers_hash_bucket_size` *size*;
Default	`proxy_headers_hash_bucket_size 64;`
Context	http, server, location

Sets the bucket size for hash tables used by the *proxy_hide_header* and *proxy_set_header* directives. The details of setting up hash tables are provided *separately*.

proxy_headers_hash_max_size

Syntax	`proxy_headers_hash_max_size` *size*;
Default	`proxy_headers_hash_max_size 512;`
Context	http, server, location

Sets the maximum size of hash tables used by the *proxy_hide_header* and *proxy_set_header* directives. The details of setting up hash tables are provided *separately*.

proxy_hide_header

Syntax	`proxy_hide_header` *field*;
Default	—
Context	http, server, location

By default, Angie does not pass the header fields "Date", "Server", "X-Pad", and "X-Accel-..." from the response of a proxied server to a client. The *proxy_hide_header* directive sets additional fields that will not be passed. If, on the contrary, the passing of fields needs to be permitted, the *proxy_pass_header* directive can be used.

proxy_http_version

Syntax	`proxy_http_version 1.0 | 1.1 | 3;`
Default	`proxy_http_version 1.0;`
Context	http, server, location, if in location, limit_except

Sets the HTTP protocol version for proxying. By default, version 1.0 is used. Version 1.1 or higher is recommended for use with *keepalive connections*.

proxy_http3_hq

Syntax	`proxy_http3_hq on | off;`
Default	`proxy_http3_hq off;`
Context	http, server

Toggles the special `hq-interop` negotiation mode, which is used for *QUIC* interop tests that Angie relies on.

Attention

Enable this mode only to run specialized tests that explicitly require it.

proxy_http3_max_concurrent_streams

Syntax	`proxy_http3_max_concurrent_streams` *number*;
Default	`proxy_http3_max_concurrent_streams 128;`
Context	http, server

Initializes HTTP/3 and QUIC settings and sets the maximum number of concurrent HTTP/3 request streams in a *connection*. Requires enabling *keepalive connections*.

proxy_http3_max_table_capacity

Syntax	`proxy_http3_max_table_capacity` *number*;
Default	`proxy_http3_max_table_capacity 4096;`
Context	http, server, location

Sets the dynamic table capacity for proxy connections.

Note

A similar *http3_max_table_capacity* directive does this for server connections. To avoid errors, dynamic table usage is disabled when proxying with caching is enabled.

proxy_http3_stream_buffer_size

Syntax	`proxy_http3_stream_buffer_size` *size*;
Default	`proxy_http3_stream_buffer_size 64k;`
Context	http, server

Sets the *size* of the read-write buffer used with *QUIC streams*.

proxy_ignore_client_abort

Syntax	`proxy_ignore_client_abort on | off;`
Default	`proxy_ignore_client_abort off;`
Context	http, server, location

Determines whether the connection with a proxied server should be closed when a client closes the connection without waiting for a response.

proxy_ignore_headers

Syntax	`proxy_ignore_headers` *field* ...;
Default	—
Context	http, server, location

Disables processing of certain response header fields from the proxied server. The following fields can be ignored: "X-Accel-Redirect", "X-Accel-Expires", "X-Accel-Limit-Rate", "X-Accel-Buffering", "X-Accel-Charset", "Expires", "Cache-Control", "Set-Cookie", and "Vary".

If not disabled, processing of these header fields has the following effect:

- "X-Accel-Expires", "Expires", "Cache-Control", "Set-Cookie" and "Vary" set the parameters of response *caching*;
- "X-Accel-Redirect" performs an *internal* redirect to the specified URI;
- "X-Accel-Limit-Rate" sets the *rate limit* for transmission of a response to a client;
- "X-Accel-Buffering" enables or disables *buffering* of a response;
- "X-Accel-Charset" sets the desired *charset* of a response.

proxy_intercept_errors

Syntax	`proxy_intercept_errors on | off;`
Default	`proxy_intercept_errors off;`
Context	http, server, location

Determines whether proxied responses with codes greater than or equal to 300 should be passed to a client or be intercepted and redirected to Angie for processing with the *error_page* directive.

proxy_limit_rate

Syntax	`proxy_limit_rate` *rate*;
Default	`proxy_limit_rate 0;`
Context	http, server, location

Limits the speed of reading the response from the proxied server. The *rate* is specified in bytes per second and can contain variables.

`0`	disables rate limiting

Note

The limit is set per a request, and so if Angie simultaneously opens two connections to the proxied server, the overall rate will be twice as much as the specified limit. The limitation works only if *buffering* of responses from the proxied server is enabled.

proxy_max_temp_file_size

Syntax	`proxy_max_temp_file_size` *size*;
Default	`proxy_max_temp_file_size 1024m;`
Context	http, server, location

When *buffering* of responses from the proxied server is enabled, and the whole response does not fit into the buffers set by the *proxy_buffer_size* and *proxy_buffers* directives, a part of the response can be saved to a temporary file. This directive sets the maximum size of the temporary file. The size of data written to the temporary file at a time is set by the *proxy_temp_file_write_size* directive.

`0`	disables buffering of responses to temporary files

Note

This restriction does not apply to responses that will be cached or *stored on disk*.

proxy_method

Syntax	`proxy_method` *method*;
Default	—
Context	http, server, location

Specifies the HTTP method to use in requests forwarded to the proxied server instead of the method from the client request. Parameter value can contain variables.

proxy_next_upstream

Syntax	`proxy_next_upstream error` | `timeout` | `invalid_header` | `http_500` | `http_502` | `http_503` | `http_504` | `http_403` | `http_404` | `http_429` | `non_idempotent` | `off` ...;
Default	`proxy_next_upstream error timeout;`
Context	http, server, location

Specifies in which cases a request should be passed to the next server in the *upstream pool*:

`error`	an error occurred while establishing a connection with the server, passing a request to it, or reading the response header;
`timeout`	a timeout has occurred while establishing a connection with the server, passing a request to it, or reading the response header;
`invalid_header`	a server returned an empty or invalid response;
`http_500`	a server returned a response with the code 500;
`http_502`	a server returned a response with the code 502;
`http_503`	a server returned a response with the code 503;
`http_504`	a server returned a response with the code 504;
`http_403`	a server returned a response with the code 403;
`http_404`	a server returned a response with the code 404;
`http_429`	a server returned a response with the code 429;
`non_idempotent`	normally, requests with a non-idempotent method (`POST`, `LOCK`, `PATCH`) are not passed to the next server if a request has been sent to an upstream server; enabling this option explicitly allows retrying such requests;
`off`	disables passing a request to the next server.

> **Note**
>
> One should bear in mind that passing a request to the next server is only possible if nothing has been sent to a client yet. That is, if an error or timeout occurs in the middle of the transferring of a response, fixing this is impossible.

The directive also defines what is considered an *unsuccessful attempt* of communication with a server.

`error`	always considered unsuccessful attempts, even if they are not specified in the directive
`timeout`	
`invalid_header`	
`http_500`	considered unsuccessful attempts only if they are specified in the directive
`http_502`	
`http_503`	
`http_504`	
`http_429`	
`http_403`	never considered unsuccessful attempts
`http_404`	

Passing a request to the next server can be limited by the *number of tries* and by *time*.

proxy_next_upstream_timeout

Syntax	`proxy_next_upstream_timeout` *time*;
Default	`proxy_next_upstream_timeout 0;`
Context	http, server, location

Limits the time during which a request can be passed to the *next* server.

`0`	turns off this limitation

proxy_next_upstream_tries

Syntax	`proxy_next_upstream_tries` *number*;
Default	`proxy_next_upstream_tries 0;`
Context	http, server, location

Limits the number of possible tries for passing a request to the *next* server.

`0`	turns off this limitation

proxy_no_cache

Syntax	`proxy_no_cache` *string* ...;
Default	—
Context	http, server, location

Defines conditions under which the response will not be saved to a cache. If at least one value of the string parameters is not empty and is not equal to "0" then the response will not be saved:

```
proxy_no_cache $cookie_nocache $arg_nocache$arg_comment;
proxy_no_cache $http_pragma    $http_authorization;
```

Can be used along with the *proxy_cache_bypass* directive.

proxy_pass

Syntax	`proxy_pass` *uri*;
Default	—
Context	location, if in location, limit_except

Sets the protocol and address of a proxied server and an optional URI to which a location should be mapped. As a protocol, `http` or `https` can be specified. The address can be specified as a domain name or IP address, and an optional port:

```
proxy_pass http://localhost:8000/uri/;
```

or as a UNIX domain socket path specified after the word `unix` and enclosed in colons:

```
proxy_pass http://unix:/tmp/backend.socket:/uri/;
```

If a domain name resolves to several addresses, all of them will be used in a round-robin fashion. In addition, an address can be specified as a *server group*.

Parameter value can contain variables. In this case, if an address is specified as a domain name, the name is searched among the described server groups, and, if not found, is determined using a *resolver*.

A request URI is passed to the server as follows:

- If the `proxy_pass` directive is specified **with a URI**, then when a request is passed to the server, the part of a *normalized* request URI matching the location is replaced by a URI specified in the directive:

```
location /name/ {
    proxy_pass http://127.0.0.1/remote/;
}
```

- If `proxy_pass` is specified **without a URI**, the request URI is passed to the server in the same form as sent by a client when the original request is processed, or the full normalized request URI is passed when processing the changed URI:

```
location /some/path/ {
    proxy_pass http://127.0.0.1;
}
```

In some cases, the part of a request URI to be replaced cannot be determined:

- When `location` is specified using a regular expression, and also inside named *locations*

In these cases, `proxy_pass` should be specified without a URI.

- When the URI is changed inside a proxied `location` using the *rewrite* directive, and this same configuration will be used to process a request (*break*):

```
location /name/ {
    rewrite    /name/([^/]+) /users?name=$1 break;
    proxy_pass http://127.0.0.1;
}
```

In this case, the URI specified in the directive is ignored and the full changed request URI is passed to the server.

- When variables are used in `proxy_pass`:

```
location /name/ {
    proxy_pass http://127.0.0.1$request_uri;
}
```

In this case, if URI is specified in the directive, it is passed to the server as is, replacing the original request URI.

WebSocket proxying requires special configuration.

proxy_pass_header

Syntax	`proxy_pass_header` *field* ...;
Default	—
Context	http, server, location

Permits passing *otherwise disabled* header fields from a proxied server to a client.

proxy_pass_request_body

Syntax	`proxy_pass_request_body on | off;`
Default	`proxy_pass_request_body on;`
Context	http, server, location

Indicates whether the original request body is passed to the proxied server.

```
location /x-accel-redirect-here/ {
    proxy_method GET;
    proxy_pass_request_body off;
    proxy_set_header Content-Length "";

    proxy_pass ...;
}
```

See also the *proxy_set_header* and *proxy_pass_request_headers* directives.

proxy_pass_request_headers

Syntax	`proxy_pass_request_headers on | off;`
Default	`proxy_pass_request_headers on;`
Context	http, server, location

Indicates whether the header fields of the original request are passed to the proxied server.

```
location /x-accel-redirect-here/ {
    proxy_method GET;
    proxy_pass_request_headers off;
    proxy_pass_request_body off;

    proxy_pass ...;
}
```

See also the *proxy_set_header* and *proxy_pass_request_body* directives.

proxy_pass_trailers

Syntax	`proxy_pass_trailers on	off;`
Default	`proxy_pass_trailers off;`
Context	http, server, location

Allows passing trailer fields from a proxied server to a client.

A trailer section in HTTP/1.1 is explicitly enabled.

```
location / {
    proxy_http_version 1.1;
    proxy_set_header Connection "te";
    proxy_set_header TE "trailers";
    proxy_pass_trailers on;

    proxy_pass ...;
}
```

proxy_quic_active_connection_id_limit

Syntax	`proxy_quic_active_connection_id_limit` *number*;
Default	`proxy_quic_active_connection_id_limit 2;`
Context	http, server

Sets the *QUIC* `active_connection_id_limit` transport parameter value. This is the maximum number of active connection IDs that can be maintained per server.

proxy_quic_gso

Syntax	`proxy_quic_gso on	off;`
Default	`proxy_quic_gso off;`
Context	http, server

Toggles sending data in *QUIC*-optimized batch mode using (generic segmentation offload).

proxy_quic_host_key

Syntax	`proxy_quic_host_key` *file*;
Default	—
Context	http, server

Sets a *file* with the secret key used with *QUIC* to encrypt Stateless Reset and Address Validation tokens. By default, a random key is generated at each restart. Tokens generated with old keys are not accepted.

proxy_read_timeout

Syntax	`proxy_read_timeout` *time*;
Default	`proxy_read_timeout 60s;`
Context	http, server, location

Defines a timeout for reading a response from the proxied server. The timeout is set only between two successive read operations, not for the transmission of the whole response. If the proxied server does not transmit anything within this time, the connection is closed.

proxy_redirect

Syntax	`proxy_redirect default;`
	`proxy_redirect off;`
	`proxy_redirect` *redirect replacement*`;`

Default	`proxy_redirect default;`
Context	http, server, location

Sets the text that should be changed in the "Location" and "Refresh" header fields of a proxied server response.

Suppose a proxied server returned the header field:

```
Location: http://localhost:8000/two/some/uri/
```

The directive

```
proxy_redirect http://localhost:8000/two/ http://frontend/one/;
```

will rewrite this string to:

```
Location: http://frontend/one/some/uri/
```

A server name may be omitted in the *replacement* string:

```
proxy_redirect http://localhost:8000/two/ /;
```

then the primary server's name and port, if different from 80, will be inserted.

The default replacement specified by the `default` parameter uses the parameters of the *location* and *proxy_pass* directives. Hence, the two configurations below are equivalent:

```
location /one/ {
    proxy_pass     http://upstream:port/two/;
    proxy_redirect default;
```

```
location /one/ {
    proxy_pass     http://upstream:port/two/;
    proxy_redirect http://upstream:port/two/ /one/;
```

Caution

The default parameter is not permitted if *proxy_pass* is specified using variables.

A *replacement* string can contain variables:

```
proxy_redirect http://localhost:8000/ http://$host:$server_port/;
```

A *redirect* can also contain variables:

```
proxy_redirect http://$proxy_host:8000/ /;
```

The directive can be specified using regular expressions. In this case, *redirect* should either start with the "~" symbol for a case-sensitive matching, or with the "~*" symbols for case-insensitive matching. The regular expression can contain named and positional captures, and *replacement* can reference them:

```
proxy_redirect ~^(http://[^:]+):\d+(/.+)$ $1$2;
proxy_redirect ~*/user/([^/]+)/(.+)$      http://$1.example.com/$2;
```

Several *proxy_redirect* directives can be specified on the same level:

```
proxy_redirect default;
proxy_redirect http://localhost:8000/  /;
proxy_redirect http://www.example.com/ /;
```

If several directives can be applied to the header fields of a proxied server response, the first matching directive will be chosen.

The `off` parameter cancels the effect of the *proxy_redirect* directives inherited from the previous configuration level.

Using this directive, it is also possible to add host names to relative redirects issued by a proxied server:

```
proxy_redirect / /;
```

proxy_request_buffering

Syntax	`proxy_request_buffering on | off;`
Default	`proxy_request_buffering on;`
Context	http, server, location

Enables or disables buffering of a client request body.

`on`	the entire request body is *read* from the client before sending the request to a proxied server.
`off`	the request body is sent to the proxied server immediately as it is received. In this case, the request cannot be passed to the *next server* if Angie already started sending the request body.

When HTTP/1.1 chunked transfer encoding is used to send the original request body, the request body will be buffered regardless of the directive value unless HTTP/1.1 is *enabled* for proxying.

proxy_send_lowat

Syntax	`proxy_send_lowat` *size*;
Default	`proxy_send_lowat 0;`
Context	http, server, location

If the directive is set to a non-zero value, Angie will try to minimize the number of send operations on outgoing connections to a proxied server by using either *NOTE_LOWAT* flag of the *kqueue* method, or the *SO_SNDLOWAT* socket option, with the specified size.

> **Note**
>
> This directive is ignored on Linux, Solaris, and Windows.

proxy_send_timeout

Syntax	`proxy_send_timeout` *time*;
Default	`proxy_send_timeout 60s;`
Context	http, server, location

Sets a timeout for transmitting a request to the proxied server. The timeout is set only between two successive write operations, not for the transmission of the whole request. If the proxied server does not receive anything within this time, the connection is closed.

proxy_set_body

Syntax	`proxy_set_body` *value*;
Default	—
Context	http, server, location

Allows redefining the request body passed to the proxied server. The value can contain text, variables, and their combination.

proxy_set_header

Syntax	`proxy_set_header` *field value*;
Default	`proxy_set_header Host $proxy_host;`
Context	http, server, location

Allows redefining or appending fields to the request header *passed* to the proxied server. The *value* can contain text, variables, and their combinations. These directives are inherited from the previous configuration level if and only if there are no *proxy_set_header* directives defined on the current level. By default, only two fields are redefined:

```
proxy_set_header Host       $proxy_host;
proxy_set_header Connection close;
```

If caching is enabled, the header fields "If-Modified-Since", "If-Unmodified-Since", "If-None-Match", "If-Match", "Range", and "If-Range" from the original request are not passed to the proxied server.

An unchanged "Host" request header field can be passed like this:

```
proxy_set_header Host       $http_host;
```

However, if this field is not present in a client request header then nothing will be passed. In such a case it is better to use the *$host* variable - its value equals the server name in the "Host" request header field or the primary server name if this field is not present:

```
proxy_set_header Host       $host;
```

In addition, the server name can be passed together with the port of the proxied server:

```
proxy_set_header Host       $host:$proxy_port;
```

If the value of a header field is an empty string then this field will not be passed to a proxied server:

```
proxy_set_header Accept-Encoding "";
```

proxy_socket_keepalive

Syntax	`proxy_socket_keepalive on | off;`
Default	`proxy_socket_keepalive off;`
Context	http, server, location

Configures the "TCP keepalive" behavior for outgoing connections to a proxied server.

`""`	By default, the operating system's settings are in effect for the socket.
`on`	The *SO_KEEPALIVE* socket option is turned on for the socket.

proxy_ssl_certificate

Syntax	`proxy_ssl_certificate` *file* [*file*];
Default	—
Context	http, server, location

Specifies a file with the certificate in the PEM format used for authentication to a proxied HTTPS server. Variables can be used in the file name.

Added in version 1.2.0.

When *proxy_ssl_ntls* enabled, directive accepts two arguments instead of one, sign and encryption parts of certificate:

```
location /proxy {
    proxy_ssl_ntls  on;

    proxy_ssl_certificate      sign.crt enc.crt;
    proxy_ssl_certificate_key  sign.key enc.key;

    proxy_ssl_ciphers "ECC-SM2-WITH-SM4-SM3:ECDHE-SM2-WITH-SM4-SM3:RSA";

    proxy_pass https://backend:443;
}
```

proxy_ssl_certificate_cache

Syntax	`proxy_ssl_certificate_cache off;` `proxy_ssl_certificate_cache max=`*N* [`inactive=`*time*] [`valid=`*time*];
Default	`proxy_ssl_certificate_cache off;`
Context	http, server, location

Defines a cache that stores *SSL certificates* and *secret keys* specified using variables.

The directive supports the following parameters:

- `max` — sets the maximum number of elements in the cache. When the cache overflows, the least recently used (LRU) elements are removed.
- `inactive` — defines the time after which an element is removed if it has not been accessed. The default is 10 seconds.
- `valid` — defines the time during which a cached element is considered valid and can be reused. The default is 60 seconds. After this period, certificates are reloaded or revalidated.

- `off` — disables the cache.

Example:

```
proxy_ssl_certificate       $proxy_ssl_server_name.crt;
proxy_ssl_certificate_key   $proxy_ssl_server_name.key;
proxy_ssl_certificate_cache max=1000 inactive=20s valid=1m;
```

proxy_ssl_certificate_key

Syntax	`proxy_ssl_certificate_key` *file* [*file*];
Default	—
Context	http, server, location

Specifies a file with the secret key in the PEM format used for authentication to a proxied HTTPS server.

The value "engine:*name*:id" can be specified instead of the file, which loads a secret key with a specified id from the OpenSSL engine name. Variables can be used in the file name.

Added in version 1.2.0.

When *proxy_ssl_ntls* enabled, directive accepts two arguments instead of one: sign and encryption parts of key:

```
location /proxy {
    proxy_ssl_ntls  on;

    proxy_ssl_certificate      sign.crt enc.crt;
    proxy_ssl_certificate_key  sign.key enc.key;

    proxy_ssl_ciphers "ECC-SM2-WITH-SM4-SM3:ECDHE-SM2-WITH-SM4-SM3:RSA";

    proxy_pass https://backend:443;
}
```

proxy_ssl_ciphers

Syntax	`proxy_ssl_ciphers` *ciphers*;
Default	`proxy_ssl_ciphers DEFAULT;`
Context	http, server, location

Specifies the enabled ciphers for requests to a proxied HTTPS server. The ciphers are specified in the format understood by the OpenSSL library.

The list of ciphers depends on the version of OpenSSL installed. The full list can be viewed using the `openssl ciphers` command.

Attention

The `proxy_ssl_ciphers` directive does *not* configure ciphers for TLS 1.3 when using OpenSSL. To tune TLS 1.3 ciphers with OpenSSL, use the *proxy_ssl_conf_command* directive, which was added to support advanced SSL configuration.

- In LibreSSL, TLS 1.3 ciphers *can* be configured using `proxy_ssl_ciphers`.
- In BoringSSL, TLS 1.3 ciphers cannot be configured at all.

proxy_ssl_conf_command

Syntax	`proxy_ssl_conf_command` *name value*;
Default	—
Context	http, server, location

Sets arbitrary OpenSSL configuration commands when establishing a connection with the proxied HTTPS server.

Important

The directive is supported when using OpenSSL 1.0.2 or higher. To configure TLS 1.3 ciphers with OpenSSL, use the `ciphersuites` command.

Several *proxy_ssl_conf_command* directives can be specified on the same level. These directives are inherited from the previous configuration level if and only if there are no *proxy_ssl_conf_command* directives defined on the current level.

Caution

Note that configuring OpenSSL directly might result in unexpected behavior.

proxy_ssl_crl

Syntax	`proxy_ssl_crl` *file*;
Default	—
Context	http, server, location

Specifies a file with revoked certificates (CRL) in the PEM format used to *verify* the certificate of the proxied HTTPS server.

proxy_ssl_name

Syntax	`proxy_ssl_name` *name*;
Default	`proxy_ssl_name $proxy_host;`
Context	http, server, location

Allows overriding the server name used to *verify* the certificate of the proxied HTTPS server and to be *passed through SNI* when establishing a connection with the proxied HTTPS server.

By default, the host part of the *proxy_pass* URL is used.

proxy_ssl_ntls

Added in version 1.2.0.

Syntax	`proxy_ssl_ntls on` | `off`;
Default	`proxy_ssl_ntls off;`
Context	http, server

Enables client-side support for NTLS using TongSuo library.

```
location /proxy {
    proxy_ssl_ntls  on;

    proxy_ssl_certificate      sign.crt enc.crt;
    proxy_ssl_certificate_key  sign.key enc.key;

    proxy_ssl_ciphers "ECC-SM2-WITH-SM4-SM3:ECDHE-SM2-WITH-SM4-SM3:RSA";

    proxy_pass https://backend:443;
}
```

Important

Build Angie using the *--with-ntls* build option and link with NTLS-enabled SSL library

```
./configure --with-openssl=../Tongsuo-8.3.0 \
            --with-openssl-opt=enable-ntls  \
            --with-ntls
```

proxy_ssl_password_file

Syntax	`proxy_ssl_password_file` *file*;
Default	—
Context	http, server, location

Specifies a file with passphrases for *secret keys* where each passphrase is specified on a separate line. Passphrases are tried in turn when loading the key.

proxy_ssl_protocols

Syntax	`proxy_ssl_protocols` [SSLv2] [SSLv3] [TLSv1] [TLSv1.1] [TLSv1.2] [TLSv1.3];
Default	`proxy_ssl_protocols TLSv1.2 TLSv1.3;`
Context	http, server, location

Changed in version 1.2.0: `TLSv1.3` parameter added to default set.

Enables the specified protocols for requests to a proxied HTTPS server.

proxy_ssl_server_name

Syntax	`proxy_ssl_server_name on` | `off`;
Default	`proxy_ssl_server_name off;`
Context	http, server, location

Enables or disables passing the server name set by the *proxy_ssl_name* directive via the Server Name Indication TLS extension (SNI, RFC 6066) while establishing a connection with the proxied HTTPS server.

proxy_ssl_session_reuse

Syntax	`proxy_ssl_session_reuse on	off;`
Default	`proxy_ssl_session_reuse on;`
Context	http, server, location

Determines whether SSL sessions can be reused when working with the proxied server. If the errors "*SSL3_GET_FINISHED:digest check failed*" appear in the logs, try disabling *session reuse*.

proxy_ssl_trusted_certificate

Syntax	`proxy_ssl_trusted_certificate` *file*;
Default	—
Context	http, server, location

Specifies a file with trusted CA certificates in the PEM format used to *verify* the certificate of the proxied HTTPS server.

proxy_ssl_verify

Syntax	`proxy_ssl_verify on	off;`
Default	`proxy_ssl_verify off;`
Context	http, server, location

Enables or disables verification of the proxied HTTPS server certificate.

proxy_ssl_verify_depth

Syntax	`proxy_ssl_verify_depth` *number*;
Default	`proxy_ssl_verify_depth 1;`
Context	http, server, location

Sets the verification depth in the proxied HTTPS server certificates chain.

proxy_store

Syntax	`proxy_store on	off |` *string*;
Default	`proxy_store off;`
Context	http, server, location

Enables saving of files to a disk.

`on`	saves files with paths corresponding to the directives *alias* or *root*
`off`	disables saving of files

The file name can be set explicitly using the `string` with variables:

```
proxy_store /data/www$original_uri;
```

The modification time of files is set according to the received "Last-Modified" response header field. The response is first written to a temporary file, and then the file is renamed. Temporary files and the persistent store can be put on different file systems. However, be aware that in this case a file is copied across two file systems instead of the cheap renaming operation. It is thus recommended that for any given *location* both saved files and a directory holding temporary files, set by the *proxy_temp_path* directive, are put on the same file system.

This directive can be used to create local copies of static unchangeable files, e.g.:

```
location /images/ {
    root               /data/www;
    error_page         404 = /fetch$uri;
}

location /fetch/ {
    internal;

    proxy_pass         http://backend/;
    proxy_store        on;
    proxy_store_access user:rw group:rw all:r;
    proxy_temp_path    /data/temp;

    alias              /data/www/;
}
```

or like this:

```
location /images/ {
    root               /data/www;
    error_page         404 = @fetch;
}

location @fetch {
    internal;

    proxy_pass         http://backend;
    proxy_store        on;
    proxy_store_access user:rw group:rw all:r;
    proxy_temp_path    /data/temp;

    root               /data/www;
}
```

proxy_store_access

Syntax	`proxy_store_access` *users:permissions ...*;
Default	`proxy_store_access user:rw;`
Context	http, server, location

Sets access permissions for newly created files and directories, e.g.:

```
proxy_store_access user:rw group:rw all:r;
```

If any `group` or `all` access permissions are specified then user permissions may be omitted:

```
proxy_store_access group:rw all:r;
```

proxy_temp_file_write_size

Syntax	`proxy_temp_file_write_size` *size*;
Default	`proxy_temp_file_write_size 8k|16k;`
Context	http, server, location

Limits the size of data written to a temporary file at a time, when buffering of responses from the proxied server to temporary files is enabled. By default, size is limited by two buffers set by the *proxy_buffer_size* and *proxy_buffers* directives. The maximum size of a temporary file is set by the *proxy_max_temp_file_size* directive.

proxy_temp_path

Syntax	`proxy_temp_path` *path* [*level1* [*level2* [*level3*]]]`;`
Default	`proxy_temp_path proxy_temp;` (the path depends on the `--http-proxy-temp-path` build option)
Context	http, server, location

Defines a directory for storing temporary files with data received from proxied servers. Up to three-level subdirectory hierarchy can be used underneath the specified directory. For example, in the following configuration

```
proxy_temp_path /spool/angie/proxy_temp 1 2;
```

a temporary file might look like this:

```
/spool/angie/proxy_temp/7/45/00000123457
```

See also the use_temp_path parameter of the *proxy_cache_path* directive.

Built-in Variables

The *http_proxy* module supports built-in variables that can be used to compose headers using the *proxy_set_header* directive:

`$proxy_host`

name and port of a proxied server as specified in the *proxy_pass* directive;

`$proxy_port`

port of a proxied server as specified in the *proxy_pass* directive, or the protocol's default port;

`$proxy_add_x_forwarded_for`

the "X-Forwarded-For" client request header field with the *$remote_addr* variable appended to it, separated by a comma. If the "X-Forwarded-For" field is not present in the client request header, the *$proxy_add_x_forwarded_for* variable is equal to the *$remote_addr* variable.

Random Index

The module processes requests ending with the slash character (`/`) and picks a random file in a directory to serve as an index file. The module is processed before the *http_index* module.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-http_random_index_module` build option.

In packages and images from our repos, the module is included in the build.

Configuration Example

```
location / {
    random_index on;
}
```

Directives

random_index

Syntax	`random_index on	off;`
Default	`random_index off;`
Context	location

Enables or disables module processing in a surrounding location.

RealIP

The module is used to change the client address and optional port to those sent in the specified header field.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-http_realip_module` build option.

In packages and images from our repos, the module is included in the build.

Configuration Example

```
set_real_ip_from  192.168.1.0/24;
set_real_ip_from  192.168.2.1;
set_real_ip_from  2001:0db8::/32;
real_ip_header    X-Forwarded-For;
real_ip_recursive on;
```

Directives

set_real_ip_from

Syntax	`set_real_ip_from` *address* | *CIDR* | `unix:;`
Default	—
Context	http, server, location

Defines trusted addresses that are known to send correct replacement addresses. If the special value `unix:` is specified, all UNIX domain sockets will be trusted. Trusted addresses may also be specified using a hostname.

real_ip_header

Syntax	`real_ip_header` *field* | `X-Real-IP` | `X-Forwarded-For` | `proxy_protocol;`
Default	`real_ip_header X-Real-IP;`
Context	http, server, location

Defines the request header field whose value will be used to replace the client address.

The request header field value that contains an optional port is also used to replace the client port. The address and port should be specified according to RFC 3986.

The `proxy_protocol` parameter changes the client address to the one from the PROXY protocol header. The PROXY protocol must be previously enabled by setting the *proxy_protocol* parameter in the *listen* directive.

real_ip_recursive

Syntax	`real_ip_recursive on | off;`
Default	`real_ip_recursive off;`
Context	http, server, location

If recursive search is disabled, the original client address that matches one of the trusted addresses is replaced by the last address sent in the request header field defined by the *real_ip_header* directive. If recursive search is enabled, the original client address that matches one of the trusted addresses is replaced by the last non-trusted address sent in the request header field.

Built-in Variables

`$realip_remote_addr`

keeps the original client address

`$realip_remote_port`

keeps the original client port

Referer

The module is used to block access to a site for requests with invalid values in the "Referer" header field. It should be kept in mind that fabricating a request with an appropriate "Referer" field value is quite easy, and so the intended purpose of this module is not to block such requests thoroughly but to block the mass flow of requests sent by regular browsers. It should also be taken into consideration that regular browsers may not send the "Referer" field even for valid requests.

Configuration Example

```
valid_referers none blocked server_names
               *.example.com example.* www.example.org/galleries/
               ~\.google\.;

if ($invalid_referer) {
    return 403;
}
```

Directives

referer_hash_bucket_size

Syntax	`referer_hash_bucket_size` *size*;
Default	`referer_hash_bucket_size 64;`
Context	server, location

Sets the bucket size for the valid referers hash tables. The details of setting up hash tables are provided in a separate *document*.

referer_hash_max_size

Syntax	`referer_hash_max_size` *size*;
Default	`referer_hash_max_size 2048;`
Context	server, location

Sets the maximum size of the valid referers hash tables. The details of setting up hash tables are provided in a separate *document*.

valid_referers

Syntax	`valid_referers none` | `blocked` | `server_names` | *string* ...;
Default	—
Context	server, location

Specifies the "Referer" request header field values that will cause the built-in *$invalid_referer* variable to be set to an empty string. Otherwise, the variable will be set to "1". Search for a match is case-insensitive.

Parameters can be as follows:

`none`	the "Referer" field is missing in the request header;
`blocked`	the "Referer" field is present in the request header, but its value has been deleted by a firewall or proxy server; such values are strings that do not start with `http://` or `https://`;
`server_names`	the "Referer" request header field contains one of the server names;
`arbitrary string`	defines a server name and an optional URI prefix. A server name can have an "*" at the beginning or end. During the checking, the server's port in the "Referer" field is ignored;
`regular expression`	the first symbol should be a "~". It should be noted that an expression will be matched against the text starting after the `http://` or `https://`.

Example:

```
valid_referers none blocked server_names
               *.example.com example.* www.example.org/galleries/
               ~\.google\.;
```

Built-in Variables

`$invalid_referer`

Empty string, if the "Referer" request header field value is considered *valid*, otherwise "1".

Rewrite

The module is used to change request URI using PCRE regular expressions, return redirects, and conditionally select configurations.

The *break*, *if*, *return*, *rewrite* and *set* directives are processed in the following order:

- the directives of this module specified on the *server* level are executed sequentially;
- repeatedly:

– a *location* is searched based on a request URI;
– the directives of this module specified inside the found location are executed sequentially;
– the loop is repeated if a request URI was *rewritten*, but *not more* than 10 times.

Directives

break

Syntax	`break;`
Default	—
Context	server, location, if

Stops processing the current set of *http_ rewrite* directives.

If a directive is specified inside the *location*, further processing of the request continues in this *location*.

Example:

```
if ($slow) {
    limit_rate 10k;
    break;
}
```

if

Syntax	`if` (*condition*) { ... }
Default	—
Context	server, location

The specified condition is evaluated. If true, this module directives specified inside the braces are executed, and the request is assigned the configuration inside the *if* directive. Configurations inside the *if* directives are inherited from the previous configuration level.

A condition may be any of the following:

- a variable name; false if the value of a variable is an empty string or "0";
- comparison of a variable with a string using the "=" and "!=" operators;
- matching of a variable against a regular expression using the "~" (for case-sensitive matching) and "~*" (for case-insensitive matching) operators. Regular expressions can contain captures that are made available for later reuse in the $1..$9 variables. Negative operators "!~" and "!~*" are also available. If a regular expression includes the "}" or ";" characters, the whole expressions should be enclosed in single or double quotes.
- checking of a file existence with the "-f" and "!-f" operators;
- checking of a directory existence with the "-d" and "!-d" operators;
- checking of a file, directory, or symbolic link existence with the "-e" and "!-e" operators;
- checking for an executable file with the "-x" and "!-x" operators.

Examples:

```
if ($http_user_agent ~ MSIE) {
    rewrite ^(.*)$ /msie/$1 break;
}

if ($http_cookie ~* "id=([^;]+)(?:;|$)") {
```

```
    set $id $1;
}

if ($request_method = POST) {
    return 405;
}

if ($slow) {
    limit_rate 10k;
}

if ($invalid_referer) {
    return 403;
}
```

Note

A value of the *$invalid_referer* Built-in variable is set by the *valid_referers* directive.

return

Syntax	`return` *code* [*text*]; `return` *code URL*; `return` *URL*;
Default	—
Context	server, location, if

Stops processing and returns the specified `code` to a client. The non-standard code 444 closes a connection without sending a response header.

It is possible to specify either a redirect URL (for codes 301, 302, 303, 307, and 308) or the response body text (for other codes). A response body text and redirect URL can contain variables. As a special case, a redirect URL can be specified as a URI local to this server, in which case the full redirect URL is formed according to the request scheme (*$scheme*) and the *server_name_in_redirect* and *port_in_redirect* directives.

In addition, a URL for temporary redirect with the code 302 can be specified as the sole parameter. Such a parameter should start with the `http://`, `https://`, or "*$scheme*" string. A URL can contain variables.

See also the *error_page* directive.

rewrite

Syntax	`rewrite` *regex replacement* [*flag*];
Default	—
Context	server, location, if

If the specified regular expression matches a request URI, URI is changed as specified in the `replacement` string. The *rewrite* directives are executed sequentially in order of their appearance in the configuration file. It is possible to terminate further processing of the directives using `flags`. If a `replacement` string starts with `http://`, `https://`, or "*$scheme*", the processing stops and the redirect is returned to a client.

An optional `flag` parameter can be one of:

`last`	stops processing the current set of *http_ rewrite*, directives and starts a search for a new *location* matching the changed URI;
`break`	stops processing the current set of *http_ rewrite* directives as with the *break* directive;
`redirect`	returns a temporary redirect with the 302 code; used if a `replacement` string does not start with `http://`, `https://` or "*$scheme*";
`permanent`	returns a permanent redirect with the 301 code.

The full redirect URL is formed according to the request scheme (*$scheme*) and the *server_ name_ in_ redirect* and *port_ in_ redirect* directives.

Example:

```
server {
#    ...
    rewrite ^(/download/.*)/media/(.*)\..*$ $1/mp3/$2.mp3 last;
    rewrite ^(/download/.*)/audio/(.*)\..*$ $1/mp3/$2.ra  last;
    return  403;
#    ...
}
```

But if these directives are put inside the "/download/" location, the `last` flag should be replaced by `break`, or otherwise Angie will make 10 cycles and return the 500 error:

```
location /download/ {
    rewrite ^(/download/.*)/media/(.*)\..*$ $1/mp3/$2.mp3 break;
    rewrite ^(/download/.*)/audio/(.*)\..*$ $1/mp3/$2.ra  break;
    return  403;
}
```

If a `replacement` string includes the new request arguments, the previous request arguments are appended after them. If this is undesired, putting a question mark at the end of a replacement string avoids having them appended, for example:

```
rewrite ^/users/(.*)$ /show?user=$1? last;
```

If a regular expression includes the "}" or ";" characters, the whole expressions should be enclosed in single or double quotes.

rewrite_log

Syntax	`rewrite_log on | off;`
Default	`rewrite_log off;`
Context	http, server, location, if

Enables or disables logging of *http_ rewrite* module directives processing results into the *error_ log* at the *notice* level.

set

Syntax	`set` *$variable value*;
Default	—
Context	server, location, if

Sets a value for the specified variable. The value can contain text, variables, and their combination.

uninitialized_variable_warn

Syntax	`uninitialized_variable_warn on | off;`
Default	`uninitialized_variable_warn on;`
Context	http, server, location, if

Controls whether warnings about uninitialized variables are logged.

Internal Implementation

The *http_rewrite* module directives are compiled at the configuration stage into internal instructions that are interpreted during request processing. An interpreter is a simple virtual stack machine.

For example, the directives

```
location /download/ {
    if ($forbidden) {
        return 403;
    }

    if ($slow) {
        limit_rate 10k;
    }

    rewrite ^/(download/.*)/media/(.*)\..*$ /$1/mp3/$2.mp3 break;
}
```

will be translated into these instructions:

```
variable $forbidden
check against zero
    return 403
    end of code
variable $slow
check against zero
match of regular expression
copy "/"
copy $1
copy "/mp3/"
copy $2
copy ".mp3"
end of regular expression
end of code
```

Note that there are no instructions for the *limit_rate* directive above as it is unrelated to the *http_rewrite* module. A separate configuration is created for the *if* block. If the condition holds true, a request is assigned this configuration where *limit_rate* equals to 10k.

The directive

```
rewrite ^/(download/.*)/media/(.*)\..*$ /$1/mp3/$2.mp3 break;
```

can be made smaller by one instruction if the first slash in the regular expression is put inside the parentheses:

```
rewrite ^(/download/.*)/media/(.*)\..*$ $1/mp3/$2.mp3 break;
```

The corresponding instructions will then look like this:

```
match of regular expression
copy $1
copy "/mp3/"
copy $2
copy ".mp3"
end of regular expression
end of code
```

SCGI

Allows passing requests to SCGI server.

Configuration Example

```
location / {
    include   scgi_params;
    scgi_pass localhost:9000;
}
```

Directives

scgi_bind

Syntax	`scgi_bind` *address* [`transparent`] | `off`;
Default	—
Context	http, server, location

Makes outgoing connections to a SCGI server originate from the specified local IP address with an optional port. Parameter value can contain variables. The special value `off` cancels the effect of the *scgi_bind* directive inherited from the previous configuration level, which allows the system to auto-assign the local IP address and port.

The `transparent` parameter allows outgoing connections to a SCGI server originate from a non-local IP address, for example, from a real IP address of a client:

```
scgi_bind $remote_addr transparent;
```

In order for this parameter to work, it is usually necessary to run Angie worker processes with the *superuser* privileges. On Linux it is not required as if the `transparent` parameter is specified, worker processes inherit the *CAP_NET_RAW* capability from the master process.

Important

It is necessary to configure kernel routing table to intercept network traffic from the SCGI server.

scgi_buffer_size

Syntax	`scgi_buffer_size` *size*;
Default	`scgi_buffer_size 4k|8k;`
Context	http, server, location

Sets the size of the buffer used for reading the first part of the response received from the SCGI server. This part usually contains a small response header. By default, the buffer size is equal to one memory

page. This is either 4K or 8K, depending on a platform. It can be made smaller, however.

scgi_buffering

Syntax	`scgi_buffering` *size*;
Default	`scgi_buffering on;`
Context	http, server, location

Enables or disables buffering of responses from the SCGI server.

`on`	Angie receives a response from the SCGI server as soon as possible, saving it into the buffers set by the *scgi_buffer_size* and *scgi_buffers* directives. If the whole response does not fit into memory, a part of it can be saved to a *temporary file* on the disk. Writing to temporary files is controlled by the *scgi_max_temp_file_size* and *scgi_temp_file_write_size* directives.
`off`	The response is passed to a client synchronously, immediately as it is received. Angie will not try to read the whole response from the SCGI server. The maximum size of the data that Angie can receive from the server at a time is set by the *scgi_buffer_size* directive.

Buffering can also be enabled or disabled by passing "yes" or "no" in the "X-Accel-Buffering" response header field. This capability can be disabled using the *scgi_ignore_headers* directive.

scgi_buffers

Syntax	`scgi_buffers` *number size*;
Default	`scgi_buffers 8 4k	8k;`
Context	http, server, location

Sets the number and size of the buffers used for reading a response from the SCGI server, for a single connection.

By default, the buffer size is equal to one memory page. This is either 4K or 8K, depending on a platform.

scgi_busy_buffers_size

Syntax	`scgi_busy_buffers_size` *size*;
Default	`scgi_busy_buffers_size 8k	16k;`
Context	http, server, location

When *buffering* of responses from the SCGI server is enabled, limits the total size of buffers that can be busy sending a response to the client while the response is not yet fully read. In the meantime, the rest of the buffers can be used for reading the response and, if needed, buffering part of the response to a temporary file.

By default, size is limited by the size of two buffers set by the *scgi_buffer_size* and *scgi_buffers* directives.

scgi_cache

Syntax	`scgi_cache` *zone* | `off`;
Default	`scgi_cache off;`
Context	http, server, location

Defines a shared memory zone used for caching. The same zone can be used in several places. Parameter value can contain variables.

`off`	disables caching inherited from the previous configuration level.

scgi_cache_background_update

Syntax	`scgi_cache_background_update on | off;`
Default	`scgi_cache_background_update off;`
Context	http, server, location

Allows starting a background subrequest to update an expired cache item, while a stale cached response is returned to the client.

Attention

Note that it is necessary to *allow* the usage of a stale cached response when it is being updated.

scgi_cache_bypass

Syntax	`scgi_cache_bypass ...;`
Default	—
Context	http, server, location

Defines conditions under which the response will not be taken from a cache. If at least one value of the string parameters is not empty and is not equal to "0" then the response will not be taken from the cache:

```
scgi_cache_bypass $cookie_nocache $arg_nocache$arg_comment;
scgi_cache_bypass $http_pragma    $http_authorization;
```

Can be used along with the *scgi_no_cache* directive.

scgi_cache_key

Syntax	`scgi_cache_key` *string*;
Default	—
Context	http, server, location

Defines a key for caching, for example

```
scgi_cache_key localhost:9000$request_uri;
```

scgi_cache_lock

Syntax	`scgi_cache_lock on | off;`
Default	`scgi_cache_lock off;`
Context	http, server, location

When enabled, only one request at a time will be allowed to populate a new cache element identified according to the *scgi_cache_key* directive by passing a request to a SCGI server. Other requests of the same cache element will either wait for a response to appear in the cache or the cache lock for this element to be released, up to the time set by the *scgi_cache_lock_timeout* directive.

scgi_cache_lock_age

Syntax	`scgi_cache_lock_age` *time*;
Default	`scgi_cache_lock_age 5s;`
Context	http, server, location

If the last request passed to the SCGI server for populating a new cache element has not completed for the specified time, one more request may be passed to the SCGI server.

scgi_cache_lock_timeout

Syntax	`scgi_cache_lock_timeout` *time*;
Default	`scgi_cache_lock_timeout 5s;`
Context	http, server, location

Sets a timeout for *scgi_cache_lock*. When the time expires, the request will be passed to the SCGI server, however, the response will not be cached.

scgi_cache_max_range_offset

Syntax	`scgi_cache_max_range_offset` *number*;
Default	—
Context	http, server, location

Sets an offset in bytes for byte-range requests. If the range is beyond the offset, the range request will be passed to the SCGI server and the response will not be cached.

scgi_cache_methods

Syntax	`scgi_cache_methods GET` | `HEAD` | `POST` ...;
Default	`scgi_cache_methods GET HEAD;`
Context	http, server, location

If the client request method is listed in this directive then the response will be cached. "GET" and "HEAD" methods are always added to the list, though it is recommended to specify them explicitly. See also the *scgi_no_cache* directive.

scgi_cache_min_uses

Syntax	`scgi_cache_min_uses` *number*;
Default	`scgi_cache_min_uses 1;`
Context	http, server, location

Sets the number of requests after which the response will be cached.

scgi_cache_path

Syntax	`scgi_cache_path` *path* [`levels=`*levels*] [`use_temp_path=on` | `off`] `keys_zone=`*name:size* [`inactive=`*time*] [`max_size=`*size*] [`min_free=`*size*] [`manager_files=`*number*] [`manager_sleep=`*time*] [`manager_threshold=`*time*] [`loader_files=`*number*] [`loader_sleep=`*time*] [`loader_threshold=`*time*];
Default	—
Context	http

Sets the path and other parameters of a cache. Cache data are stored in files. The file name in a cache is a result of applying the MD5 function to the *cache key*.

The `levels` parameter defines hierarchy levels of a cache: from 1 to 3, each level accepts values 1 or 2. For example, in the following configuration:

```
scgi_cache_path /data/angie/cache levels=1:2 keys_zone=one:10m;
```

file names in a cache will look like this:

```
/data/angie/cache/c/29/b7f54b2df7773722d382f4809d65029c
```

A cached response is first written to a temporary file, and then the file is renamed. Temporary files and the cache can be put on different file systems. However, be aware that in this case a file is copied across two file systems instead of the cheap renaming operation. It is thus recommended that for any given location both cache and a directory holding temporary files are put on the same file system.

The directory for temporary files is set based on the `use_temp_path` parameter.

`on`	If this parameter is omitted or set to the value on, the directory set by the *scgi_temp_path* directive for the given *location* will be used.
`off`	Temporary files will be put directly in the cache directory.

In addition, all active keys and information about data are stored in a shared memory zone, whose name and size are configured by the `keys_zone` parameter. One megabyte zone can store about 8 thousand keys.

Cached data that are not accessed during the time specified by the `inactive` parameter get removed from the cache regardless of their freshness.

By default, *inactive* is set to 10 minutes.

A special **cache manager** process monitors the maximum cache size and the minimum amount of free space on the file system with cache and when the size is exceeded or there is not enough free space, it removes the least recently used data. The data is removed in iterations.

`max_size`	maximum cache size
`min_free`	minimum amount of free space on the file system with cache
`manager_files`	limits the number of items to be deleted during one iteration By default, `100`
`manager_threshol`	limits the duration of one iteration By default, `200` milliseconds
`manager_sleep`	configures a pause between iterations By default, `50` milliseconds

A minute after Angie starts, the special **cache loader** process is activated. It loads information about previously cached data stored on file system into a cache zone. The loading is also done in iterations.

`loader_files`	limits the number of items to load during one iteration By default, `100`
`loader_threshold`	limits the duration of one iteration By default, `200` milliseconds
`loader_sleep`	configures a pause between iterations By default, `50` milliseconds

scgi_cache_revalidate

Syntax	`scgi_cache_revalidate on | off;`
Default	`scgi_cache_revalidate off;`
Context	http, server, location

Enables revalidation of expired cache items using conditional requests with the "If-Modified-Since" and "If-None-Match" header fields.

scgi_cache_use_stale

Syntax	`scgi_cache_use_stale error | timeout | invalid_header | updating | http_500 | http_502 | http_503 | http_504 | http_403 | http_404 | http_429 | off ...;`
Default	`scgi_cache_use_stale off;`
Context	http, server, location

Determines in which cases a stale cached response can be used during communication with the SCGI server. The directive's parameters match the parameters of the *scgi_next_upstream* directive.

`error`	permits using a stale cached response if a SCGI server to process a request cannot be selected.
`updating`	additional parameter, permits using a stale cached response if it is currently being updated. This allows minimizing the number of accesses to SCGI servers when updating cached data.

Using a stale cached response can also be enabled directly in the response header for a specified number of seconds after the response became stale:

- The stale-while-revalidate extension of the "Cache-Control" header field permits using a stale cached response if it is currently being updated.
- The stale-if-error extension of the "Cache-Control" header field permits using a stale cached response in case of an error.

Note

This has lower priority than using the directive parameters.

To minimize the number of accesses to SCGI servers when populating a new cache element, the *scgi_cache_lock* directive can be used.

scgi_cache_valid

Syntax	`scgi_cache_valid` [*code* ...] *time*;
Default	—
Context	http, server, location

Sets caching time for different response codes. For example, the following directives

```
scgi_cache_valid 200 302 10m;
scgi_cache_valid 404      1m;
```

set 10 minutes of caching for responses with codes 200 and 302 and 1 minute for responses with code 404.

If only caching time is specified

```
scgi_cache_valid 5m;
```

then only 200, 301, and 302 responses are cached.

In addition, the `any` parameter can be specified to cache any responses:

```
scgi_cache_valid 200 302 10m;
scgi_cache_valid 301      1h;
scgi_cache_valid any      1m;
```

Note

Parameters of caching can also be set directly in the response header. This has higher priority than setting of caching time using the directive

- The "X-Accel-Expires" header field sets caching time of a response in seconds. The zero value disables caching for a response. If the value starts with the @ prefix, it sets an absolute time in seconds since Epoch, up to which the response may be cached.
- If the header does not include the "X-Accel-Expires" field, parameters of caching may be set in the header fields "Expires" or "Cache-Control".
- If the header includes the "Set-Cookie" field, such a response will not be cached.
- If the header includes the "Vary" field with the special value "*", such a response will not be cached. If the header includes the "Vary" field with another value, such a response will be cached taking into account the corresponding request header fields.

Processing of one or more of these response header fields can be disabled using the *scgi_ignore_headers* directive.

scgi_connect_timeout

Syntax	`scgi_connect_timeout` *time*;
Default	`scgi_connect_timeout 60s;`
Context	http, server, location

Defines a timeout for establishing a connection with a SCGI server. It should be noted that this timeout cannot usually exceed 75 seconds.

scgi_connection_drop

Syntax	`scgi_connection_drop` *time* | `on` | `off`;
Default	`scgi_connection_drop off;`
Context	http, server, location

Enables termination of all connections to the proxied server after it has been removed from the group or marked as permanently unavailable by a *reresolve* process or the *API command* `DELETE`.

A connection is terminated when the next read or write event is processed for either the client or the proxied server.

Setting *time* enables a connection termination *timeout*; with `on` set, connections are dropped immediately.

scgi_force_ranges

Syntax	`scgi_force_ranges off`;
Default	`scgi_force_ranges off;`
Context	http, server, location

Enables byte-range support for both cached and uncached responses from the SCGI server regardless of the "Accept-Ranges" field in these responses.

scgi_hide_header

Syntax	`scgi_hide_header` *field*;
Default	—
Context	http, server, location

By default, Angie does not pass the header fields "Status" and "X-Accel-..." from the response of a SCGI server to a client. The *scgi_hide_header* directive sets additional fields that will not be passed. If, on the contrary, the passing of fields needs to be permitted, the *scgi_pass_header* directive can be used.

scgi_ignore_client_abort

Syntax	`scgi_ignore_client_abort on` | `off`;
Default	`scgi_ignore_client_abort off;`
Context	http, server, location

Determines whether the connection with a SCGI server should be closed when a client closes the connection without waiting for a response.

scgi_ignore_headers

Syntax	`scgi_ignore_headers` *field* ...;
Default	—
Context	http, server, location

Disables processing of certain response header fields from the SCGI server. The following fields can be ignored: "X-Accel-Redirect", "X-Accel-Expires", "X-Accel-Limit-Rate", "X-Accel-Buffering", "X-Accel-Charset", "Expires", "Cache-Control", "Set-Cookie", and "Vary".

If not disabled, processing of these header fields has the following effect:

- "X-Accel-Expires", "Expires", "Cache-Control", "Set-Cookie" and "Vary" set the parameters of response *caching*;
- "X-Accel-Redirect" performs an *internal* redirect to the specified URI;
- "X-Accel-Limit-Rate" sets the *rate limit* for transmission of a response to a client;
- "X-Accel-Buffering" enables or disables *buffering* of a response;
- "X-Accel-Charset" sets the desired *charset* of a response.

scgi_intercept_errors

Syntax	`scgi_intercept_errors on | off;`
Default	`scgi_intercept_errors off;`
Context	http, server, location

Determines whether SCGI-responses with codes greater than or equal to 300 should be passed to a client or be intercepted and redirected to Angie for processing with the *error_page* directive.

scgi_limit_rate

Syntax	`scgi_limit_rate` *rate*;
Default	`scgi_limit_rate 0;`
Context	http, server, location

Limits the speed of reading the response from the SCGI server. The *rate* is specified in bytes per second and can contain variables.

0	disables rate limiting

Note

The limit is set per a request, and so if Angie simultaneously opens two connections to the SCGI server, the overall rate will be twice as much as the specified limit. The limitation works only if *buffering* of responses from the SCGI server is enabled.

scgi_max_temp_file_size

Syntax	`scgi_max_temp_file_size` *size*;
Default	`scgi_max_temp_file_size 1024m;`
Context	http, server, location

When *buffering* of responses from the SCGI server is enabled, and the whole response does not fit into the buffers set by the *scgi_buffer_size* and *scgi_buffers* directives, a part of the response can be saved to a temporary file. This directive sets the maximum size of the temporary file. The size of data written to the temporary file at a time is set by the *scgi_temp_file_write_size* directive.

0	disables buffering of responses to temporary files

> **Note**
>
> This restriction does not apply to responses that will be cached or *stored on disk*.

scgi_next_upstream

Syntax	`scgi_next_upstream error | timeout | invalid_header | http_500 | http_503 | http_403 | http_404 | http_429 | non_idempotent | off ...;`
Default	`scgi_next_upstream error timeout;`
Context	http, server, location

Specifies in which cases a request should be passed to the next server in the *upstream pool*:

`error`	an error occurred while establishing a connection with the server, passing a request to it, or reading the response header;
`timeout`	a timeout has occurred while establishing a connection with the server, passing a request to it, or reading the response header;
`invalid_header`	a server returned an empty or invalid response;
`http_500`	a server returned a response with the code 500;
`http_503`	a server returned a response with the code 503;
`http_403`	a server returned a response with the code 403;
`http_404`	a server returned a response with the code 404;
`http_429`	a server returned a response with the code 429;
`non_idempotent`	normally, requests with a non-idempotent method (`POST`, `LOCK`, `PATCH`) are not passed to the next server if a request has been sent to an upstream server; enabling this option explicitly allows retrying such requests;
`off`	disables passing a request to the next server.

> **Note**
>
> One should bear in mind that passing a request to the next server is only possible if nothing has been sent to a client yet. That is, if an error or timeout occurs in the middle of the transferring of a response, fixing this is impossible.

The directive also defines what is considered an *unsuccessful attempt* of communication with a server.

`error`
`timeout`
`invalid_header`
	always considered unsuccessful attempts, even if they are not specified in the directive

`http_500`
`http_503`
`http_429`
	considered unsuccessful attempts only if they are specified in the directive

`http_403` `http_404`	never considered unsuccessful attempts

Passing a request to the next server can be limited by the *number of tries* and by *time*.

scgi_next_upstream_timeout

Syntax	`scgi_next_upstream_timeout` *time*;
Default	`scgi_next_upstream_timeout 0;`
Context	http, server, location

Limits the time during which a request can be passed to the *next* server.

`0`	turns off this limitation

scgi_next_upstream_tries

Syntax	`scgi_next_upstream_tries` *number*;
Default	`scgi_next_upstream_tries 0;`
Context	http, server, location

Limits the number of possible tries for passing a request to the *next* server.

`0`	turns off this limitation

scgi_no_cache

Syntax	`scgi_no_cache` *string* ...;
Default	—
Context	http, server, location

Defines conditions under which the response will not be saved to a cache. If at least one value of the string parameters is not empty and is not equal to "0" then the response will not be saved:

```
scgi_no_cache $cookie_nocache $arg_nocache$arg_comment;
scgi_no_cache $http_pragma    $http_authorization;
```

Can be used along with the *scgi_cache_bypass* directive.

scgi_param

Syntax	`scgi_param` *parameter value* [`if_not_empty`];
Default	—
Context	http, server, location

Sets a parameter that should be passed to the SCGI server. The value can contain text, variables, and their combination. These directives are inherited from the previous configuration level if and only if there are no scgi_param directives defined on the current level.

Standard CGI environment variables should be provided as SCGI headers, see the scgi_params file provided in the distribution:

```
location / {
    include scgi_params;
#    ...
}
```

If the directive is specified with `if_not_empty` then such a parameter will be passed to the server only if its value is not empty:

```
scgi_param HTTPS $https if_not_empty;
```

scgi_pass

Syntax	`scgi_pass` *uri*;
Default	—
Context	location, if in location, limit_except

Sets the address of an SCGI server. The address can be specified as a domain name or IP address, and an optional port:

```
scgi_pass localhost:9000;
```

or as a UNIX domain socket path specified after the word `unix`:

```
scgi_pass unix:/tmp/scgi.socket;
```

If a domain name resolves to several addresses, all of them will be used in a round-robin fashion. In addition, an address can be specified as a *server group*. If a group is used, you cannot specify the port with it; instead, specify the port for each server within the group individually.

Parameter value can contain variables. In this case, if an address is specified as a domain name, the name is searched among the described server groups, and, if not found, is determined using a *resolver*.

scgi_pass_header

Syntax	`scgi_pass_header` *field* ...;
Default	—
Context	http, server, location

Permits passing *otherwise disabled* header fields from a SCGI server to a client.

scgi_pass_request_body

Syntax	`scgi_pass_request_body on | off;`
Default	`scgi_pass_request_body on;`
Context	http, server, location

Indicates whether the original request body is passed to the SCGI server. See also the *scgi_pass_request_headers* directive.

scgi_pass_request_headers

Syntax	`scgi_pass_request_headers on | off;`
Default	`scgi_pass_request_headers on;`
Context	http, server, location

Indicates whether the header fields of the original request are passed to the SCGI server. See also the *scgi_pass_request_body* directive.

scgi_read_timeout

Syntax	`scgi_read_timeout` *time*;
Default	`scgi_read_timeout 60s;`
Context	http, server, location

Defines a timeout for reading a response from the SCGI server. The timeout is set only between two successive read operations, not for the transmission of the whole response. If the SCGI server does not transmit anything within this time, the connection is closed.

scgi_request_buffering

Syntax	`scgi_request_buffering on` | `off`;
Default	`scgi_request_buffering on;`
Context	http, server, location

Enables or disables buffering of a client request body.

`on`	the entire request body is *read* from the client before sending the request to a SCGI server.
`off`	the request body is sent to the SCGI server immediately as it is received. In this case, the request cannot be passed to the *next server* if Angie already started sending the request body.

When HTTP/1.1 chunked transfer encoding is used to send the original request body, the request body will be buffered regardless of the directive value.

scgi_send_timeout

Syntax	`scgi_send_timeout` *time*;
Default	`scgi_send_timeout 60s;`
Context	http, server, location

Sets a timeout for transmitting a request to the SCGI server. The timeout is set only between two successive write operations, not for the transmission of the whole request. If the SCGI server does not receive anything within this time, the connection is closed.

scgi_socket_keepalive

Syntax	`scgi_socket_keepalive on` | `off`;
Default	`scgi_socket_keepalive off;`
Context	http, server, location

Configures the "TCP keepalive" behavior for outgoing connections to a SCGI server.

`""`	By default, the operating system's settings are in effect for the socket.
`on`	The *SO_KEEPALIVE* socket option is turned on for the socket.

scgi_store

Syntax	`scgi_store on` | `off` | *string*;
Default	`scgi_store off;`
Context	http, server, location

Enables saving of files to a disk.

`on`	saves files with paths corresponding to the directives *alias* or *root*
`off`	disables saving of files

The file name can be set explicitly using the `string` with variables:

```
scgi_store /data/www$original_uri;
```

The modification time of files is set according to the received "Last-Modified" response header field. The response is first written to a temporary file, and then the file is renamed. Temporary files and the persistent store can be put on different file systems. However, be aware that in this case a file is copied across two file systems instead of the cheap renaming operation. It is thus recommended that for any given *location* both saved files and a directory holding temporary files, set by the *scgi_temp_path* directive, are put on the same file system.

This directive can be used to create local copies of static unchangeable files, e.g.:

```
location /images/ {
    root               /data/www;
    error_page         404 = /fetch$uri;
}

location /fetch/ {
    internal;

    scgi_pass          backend:9000;
    ...

    scgi_store         on;
    scgi_store_access  user:rw group:rw all:r;
    scgi_temp_path     /data/temp;

    alias              /data/www/;
}
```

scgi_store_access

Syntax	`scgi_store_access` *users:permissions ...*;
Default	`scgi_store_access user:rw;`
Context	http, server, location

Sets access permissions for newly created files and directories, e.g.:

```
scgi_store_access user:rw group:rw all:r;
```

If any `group` or `all` access permissions are specified then user permissions may be omitted:

```
scgi_store_access group:rw all:r;
```

scgi_temp_file_write_size

Syntax	`scgi_temp_file_write_size` *size*;
Default	`scgi_temp_file_write_size 8k|16k;`
Context	http, server, location

Limits the size of data written to a temporary file at a time, when buffering of responses from the SCGI server to temporary files is enabled. By default, size is limited by two buffers set by the *scgi_buffer_size* and *scgi_buffers* directives. The maximum size of a temporary file is set by the *scgi_max_temp_file_size* directive.

scgi_temp_path

Syntax	`scgi_temp_path` *path* [*level1* [*level2* [*level3*]]]`;`
Default	`scgi_temp_path scgi_temp;` (the path depends on the `--http-scgi-temp-path` build option)
Context	http, server, location

Defines a directory for storing temporary files with data received from SCGI servers. Up to three-level subdirectory hierarchy can be used underneath the specified directory. For example, in the following configuration

```
scgi_temp_path /spool/angie/scgi_temp 1 2;
```

a temporary file might look like this:

```
/spool/angie/scgi_temp/7/45/00000123457
```

See also the use_temp_path parameter of the *scgi_cache_path* directive.

Secure Link

The module is used to check authenticity of requested links, protect resources from unauthorized access, and limit link lifetime.

The authenticity of a requested link is verified by comparing the checksum value passed in a request with the value computed for the request. If a link has a limited lifetime and the time has expired, the link is considered outdated. The status of these checks is made available in the *$secure_link* variable.

The module provides two alternative operation modes. The first mode is enabled by the *secure_link_secret* directive and is used to check authenticity of requested links as well as protect resources from unauthorized access. The second mode is enabled by the *secure_link* and *secure_link_md5* directives and is also used to limit lifetime of links.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-http_secure_link_module` build option.

In packages and images from our repos, the module is included in the build.

Directives

secure_link

Syntax	`secure_link` *expression*;
Default	—
Context	http, server, location

Defines a string with variables from which the checksum value and lifetime of a link will be extracted.

Variables used in an expression are usually associated with a request; see *example* below.

The checksum value extracted from the string is compared with the MD5 hash value of the expression defined by the *secure_link_md5* directive.

If the checksums are different, the *$secure_link* variable is set to an empty string. If the checksums are the same, the link lifetime is checked.

If the link has a limited lifetime and the time has expired, the *$secure_link* variable is set to `0`. Otherwise, it is set to `1`. The MD5 hash value passed in a request is encoded in base64url.

If a link has a limited lifetime, the expiration time is set in seconds since Epoch (Thu, 01 Jan 1970 00:00:00 GMT). The value is specified in the expression after the MD5 hash, and is separated by a comma. The expiration time passed in a request is available through the *$secure_link_expires* variable for a use in the *secure_link_md5* directive. If the expiration time is not specified, a link has the unlimited lifetime.

secure_link_md5

Syntax	`secure_link_md5` *expression*;
Default	—
Context	http, server, location

Defines an expression for which the MD5 hash value will be computed and compared with the value passed in a request.

The expression should contain the secured part of a link (resource) and a secret ingredient. If the link has a limited lifetime, the expression should also contain *$secure_link_expires*.

To prevent unauthorized access, the expression may contain some information about the client, such as its address and browser version.

Example:

```
location /s/ {
    secure_link $arg_md5,$arg_expires;
    secure_link_md5 "$secure_link_expires$uri$remote_addr secret";

    if ($secure_link = "") {
        return 403;
    }

    if ($secure_link = "0") {
        return 410;
    }

#    ...
}
```

The "/s/link?md5=_e4Nc3iduzkWRm01TBBNYw&expires=2147483647" link restricts access to "/s/link" for the client with the IP address 127.0.0.1. The link also has the limited lifetime until January 19, 2038 (GMT).

On UNIX, the md5 request argument value can be obtained as:

```
echo -n '2147483647/s/link127.0.0.1 secret' | \
   openssl md5 -binary | openssl base64 | tr +/ -_ | tr -d =
```

secure_link_secret

Syntax	`secure_link_secret` *word*;
Default	—
Context	location

Defines a secret word used to check authenticity of requested links.

The full URI of a requested link looks as follows:

```
/prefix/hash/link
```

where hash is a hexadecimal representation of the MD5 hash computed for the concatenation of the link and secret word, and prefix is an arbitrary string without slashes.

If the requested link passes the authenticity check, the *$secure_link* variable is set to the link extracted from the request URI. Otherwise, the *$secure_link* variable is set to an empty string.

Example:

```
location /p/ {
    secure_link_secret secret;

    if ($secure_link = "") {
        return 403;
    }

    rewrite ^ /secure/$secure_link;
}

location /secure/ {
    internal;
}
```

A request of "/p/5e814704a28d9bc1914ff19fa0c4a00a/link" will be internally redirected to "/secure/link".

On UNIX, the hash value for this example can be obtained as:

```
echo -n 'linksecret' | openssl md5 -hex
```

Built-in Variables

`$secure_link`

The status of a link check. The specific value depends on the selected operation mode.

`$secure_link_expires`

The lifetime of a link passed in a request; intended to be used only in the *secure_link_md5* directive.

Slice

The module is a filter that splits a request into subrequests, each returning a certain range of response. The filter provides more effective caching of big responses.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-http_slice_module` build option.

In packages and images from our repos, the module is included in the build.

Configuration Example

```
location / {
    slice             1m;
    proxy_cache       cache;
    proxy_cache_key   $uri$is_args$args$slice_range;
    proxy_set_header  Range $slice_range;
    proxy_cache_valid 200 206 1h;
    proxy_pass        http://localhost:8000;
}
```

In this example, the response is split into 1-megabyte cacheable slices.

Directives

slice

Syntax	`slice` *size*;
Default	`slice 0;`
Context	http, server, location

Sets the size of the slice. The zero value disables splitting responses into slices.

Warning

Note that a too low value may result in excessive memory usage and opening a large number of files.

In order for a subrequest to return the required range, the *$slice_range* variable should be passed to the proxied server as the "Range" request header field. If *caching* is enabled, *$slice_range* should be added to the *cache key* and caching of responses with 206 status code should be *enabled*.

Built-in Variables

`$slice_range`

The current slice range in HTTP byte range format, for example, `bytes=0-1048575`.

Split Clients

The module emits variables for A/B testing, canary releases, or other scenarios that require routing a percentage of clients to one server or configuration while directing the rest elsewhere.

Configuration Example

```
http {
    split_clients "${remote_addr}AAA" $variant {
                  0.5%               .one;
                  2.0%               .two;
                  *                  "";
    }

    server {
        location / {
            index index${variant}.html;
```

Directives

split_clients

Syntax	`split_clients` *string* *$variable* { ... }
Default	—
Context	http

Creates a *$variable* by hashing the *string*; the variables in *string* are substituted, the result is hashed, then the hash is mapped to the *$variable*'s string value.

The hash function uses MurmurHash2 (32-bit), and its entire value range (0 to 4294967295) is mapped to buckets in order of appearance; the percentages determine the size of the buckets. A wildcard (`*`) may occur last; hashes that fall outside other buckets are mapped to its assigned value.

An example:

```
split_clients "${remote_addr}AAA" $variant {
               0.5%               .one;
               2.0%               .two;
               *                  "";
}
```

Here, the hashed values of the interpolated `$remote_addrAAA` string are distributed as follows:

- values 0 to 21474835 (a sample of 0.5%) yield `.one`
- values 21474836 to 107374180 (a sample of 2%) yield `.two`
- values 107374181 to 4294967295 (all other values) yield `""` (an empty string)

SSI

The module is a filter that processes SSI (Server Side Includes) commands in responses passing through it.

Configuration Example

```
location / {
    ssi on;
#    ...
}
```

Directives

ssi

Syntax	`ssi on | off;`
Default	`ssi off;`
Context	http, server, location, if in location

Enables or disables processing of SSI commands in responses.

ssi_last_modified

Syntax	`ssi_last_modified on | off;`
Default	`ssi_last_modified off;`
Context	http, server, location

Allows preserving the "Last-Modified" header field from the original response during SSI processing to facilitate response caching.

By default, the header field is removed as contents of the response are modified during processing and may contain dynamically generated elements or parts that are changed independently of the original response.

ssi_min_file_chunk

Syntax	`ssi_min_file_chunk` *size*;
Default	`ssi_min_file_chunk 1k;`
Context	http, server, location

Sets the minimum size for parts of a response stored on disk, starting from which it makes sense to send them using *sendfile*.

ssi_silent_errors

Syntax	`ssi_silent_errors on | off;`
Default	`ssi_silent_errors off;`
Context	http, server, location

If enabled, suppresses the output of the "*[an error occurred while processing the directive]*" string if an error occurred during SSI processing.

ssi_types

Syntax	`ssi_types` mime-*type* ...;
Default	`ssi_types text/html;`
Context	http, server, location

Enables processing of SSI commands in responses with the specified MIME types in addition to `text/html`. The special value "*" matches any MIME type.

ssi_value_length

Syntax	`ssi_value_length` *length*;
Default	`ssi_value_length 256;`
Context	http, server, location

Sets the maximum length of parameter values in SSI commands.

SSI Commands

SSI commands have the following generic format:

```
<!--# command parameter1=value1 parameter2=value2 ... -->
```

The following commands are supported:

`block`

Defines a block that can be used as a stub in the *include* command. The block can contain other SSI commands. The command has the following parameter:

`name`

block name.

Example:

```
<!--# block name="one" -->
stub
<!--# endblock -->
```

`config`

Sets some parameters used during SSI processing, namely:

`errmsg`

a string that is output if an error occurs during SSI processing. By default, the following string is output:

[an error occurred while processing the directive]

`timefmt`

a format string passed to the strftime() function used to output date and time. By default, the following format is used:

"%A, %d-%b-%Y %H:%M:%S %Z"

The "%s" format is suitable to output time in seconds.

`echo`

Outputs the value of a variable. The command has the following parameters:

`var`

the variable name.

`encoding`

the encoding method. Possible values include `none`, `url`, and `entity`.

By default, *entity* is used.

`default`

a non-standard parameter that sets a string to be output if a variable is undefined. By default, `(none)` is output. The command

```
<!--# echo var="name" default="no" -->
```

replaces the following sequence of commands:

```
<!--# if expr="$name" --><!--# echo var="name" --><!--#
     else -->no<!--# endif -->
```

`if`

Performs a conditional inclusion. The following commands are supported:

```
<!--# if expr="..." -->
...
<!--# elif expr="..." -->
...
<!--# else -->
...
<!--# endif -->
```

Only one level of nesting is currently supported. The command has the following parameter:

`expr`

expression. An expression can be:

- variable existence check:

```
<!--# if expr="$name" -->
```

- comparison of a variable with a text:

```
<!--# if expr="$name = text" -->
<!--# if expr="$name != text" -->
```

- comparison of a variable with a regular expression:

```
<!--# if expr="$name = /text/" -->
<!--# if expr="$name != /text/" -->
```

If a *text* contains variables, their values are substituted. A regular expression can contain positional and named captures that can later be used through variables, for example:

```
<!--# if expr="$name = /(.+)@(?P<domain>.+)/" -->
  <!--# echo var="1" -->
  <!--# echo var="domain" -->
<!--# endif -->
```

`include`

Includes the result of another request into a response. The command has the following parameters:

`file`

specifies an included file, for example:

```
<!--# include file="footer.html" -->
```

`virtual`

specifies an included request, for example:

```
<!--# include virtual="/remote/body.php?argument=value" -->
```

Several requests specified on one page and processed by proxied or FastCGI/uwsgi/SCGI/gRPC servers run in parallel. If sequential processing is desired, the *wait* parameter should be used.

`stub`

a non-standard parameter that names the block whose content will be output if the included request results in an empty body or if an error occurs during the request processing, for example:

```
<!--# block name="one" --> <!--# endblock -->
<!--# include virtual="/remote/body.php?argument=value" stub="one" -->
```

The replacement block content is processed in the included request context.

`wait`

a non-standard parameter that instructs to wait for a request to fully complete before continuing with SSI processing, for example:

```
<!--# include virtual="/remote/body.php?argument=value" wait="yes" -->
```

`set`

a non-standard parameter that instructs to write a successful result of request processing to the specified variable, for example:

```
<!--# include virtual="/remote/body.php?argument=value" set="one" -->
```

The maximum size of the response is set by the *subrequest_output_buffer_size* directive:

```
location /remote/ {
    subrequest_output_buffer_size 64k;
#    ...
}
```

`set`

Sets a value of a variable. The command has the following parameters:

`var`

the variable name.

`value`

the variable value. If an assigned value contains variables, their values are substituted.

Built-in Variables

`$date_local`

current time in the local time zone. The format is set by the *config* command with the *timefmt* parameter.

`$date_gmt`

current time in GMT. The format is set by the *config* command with the *timefmt* parameter.

SSL

Provides the necessary support for HTTPS.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-http_ssl_module` build option.

In packages and images from our repos, the module is included in the build.

Important

This module requires the OpenSSL library.

Configuration Example

To reduce the processor load it is recommended to

- set the number of *worker processes* equal to the number of processors,
- enable *keep-alive* connections,
- enable the *shared* session cache,
- disable the *built-in* session cache,
- and possibly increase the session *lifetime* (by default, 5 minutes):

```
worker_processes auto;

http {

    # ...

    server {
        listen              443 ssl;
        keepalive_timeout   70;

        ssl_protocols       TLSv1.2 TLSv1.3;
        ssl_ciphers         AES128-SHA:AES256-SHA:RC4-SHA:DES-CBC3-SHA:RC4-MD5;
        ssl_certificate     /usr/local/angie/conf/cert.pem;
        ssl_certificate_key /usr/local/angie/conf/cert.key;
        ssl_session_cache   shared:SSL:10m;
        ssl_session_timeout 10m;
```

```
        # ...
    }
```

Directives

ssl_buffer_size

Syntax	`ssl_buffer_size` *size*;
Default	`ssl_buffer_size 16k;`
Context	http, server

Sets the size of the buffer used for sending data.

By default, the buffer size is 16k, which corresponds to minimal overhead when sending big responses. To minimize Time To First Byte it may be beneficial to use smaller values, for example:

```
ssl_buffer_size 4k;
```

ssl_certificate

Syntax	`ssl_certificate` *file*;
Default	—
Context	http, server

Specifies a file with the certificate in the PEM format for the given virtual server. If intermediate certificates should be specified in addition to a primary certificate, they should be specified in the same file in the following order: the primary certificate comes first, then the intermediate certificates. A secret key in the PEM format may be placed in the same file.

This directive can be specified multiple times to load certificates of different types, for example, RSA and ECDSA:

```
server {
    listen              443 ssl;
    server_name         example.com;

    ssl_certificate     example.com.rsa.crt;
    ssl_certificate_key example.com.rsa.key;

    ssl_certificate     example.com.ecdsa.crt;
    ssl_certificate_key example.com.ecdsa.key;

    # ...
}
```

Only OpenSSL 1.0.2 or higher supports separate certificate chains for different certificates. With older versions, only one certificate chain can be used.

Important

Variables can be used in the file name when using OpenSSL 1.0.2 or higher:

```
ssl_certificate     $ssl_server_name.crt;
ssl_certificate_key $ssl_server_name.key;
```

Note that using variables implies that a certificate will be loaded for each SSL handshake, and this may have a negative impact on performance.

The value `data:$variable` can be specified instead of the `file`, which loads a certificate from a variable without using intermediate files. Note that inappropriate use of this syntax may have its security implications, such as writing secret key data to *error log*.

Important

It should be kept in mind that due to the HTTPS protocol limitations for maximum interoperability virtual servers should listen on *different IP addresses*.

Added in version 1.2.0: If *ssl_ntls* is enabled, the directive can accept two arguments (the signature and the encryption parts of the key) instead of one:

```
listen ... ssl;

ssl_ntls  on;

# dual NTLS certificate
ssl_certificate      sign.crt enc.crt;
ssl_certificate_key  sign.key enc.key;

# can be used together with a regular RSA certificate
ssl_certificate  rsa.crt;
ssl_certificate  rsa.key;
```

ssl_certificate_cache

Syntax	`ssl_certificate_cache off;` `ssl_certificate_cache max=`*N* [`inactive=`*time*] [`valid=`*time*];
Default	`ssl_certificate_cache off;`
Context	http, server

Defines a cache that stores *SSL certificates* and *secret keys* specified using variables.

The directive supports the following parameters:

- `max` — sets the maximum number of elements in the cache. When the cache overflows, the least recently used (LRU) elements are removed.
- `inactive` — defines the time after which an element is removed if it has not been accessed. The default is 10 seconds.
- `valid` — defines the time during which a cached element is considered valid and can be reused. The default is 60 seconds. After this period, certificates are reloaded or revalidated.
- `off` — disables the cache.

Example:

```
ssl_certificate       $ssl_server_name.crt;
ssl_certificate_key   $ssl_server_name.key;
ssl_certificate_cache max=1000 inactive=20s valid=1m;
```

ssl_certificate_key

Syntax	`ssl_certificate_key` *file*;
Default	—
Context	http, server

Specifies a file with the secret key in the PEM format for the given virtual server.

Important

Variables can be used in the file name when using OpenSSL 1.0.2 or higher.

The value `engine:name:id` can be specified instead of the *file*, which loads a secret key with a specified *id* from the OpenSSL engine name.

The value `data:$variable` can be specified instead of the *file*, which loads a secret key from a variable without using intermediate files. Note that inappropriate use of this syntax may have its security implications, such as writing secret key data to *error log*.

Added in version 1.2.0: If *ssl_ntls* is enabled, the directive can accept two arguments (the signature and the encryption parts of the key) instead of one:

```
listen ... ssl;

ssl_ntls  on;

# dual NTLS certificate
ssl_certificate      sign.crt enc.crt;
ssl_certificate_key  sign.key enc.key;

# can be used together with a regular RSA certificate
ssl_certificate  rsa.crt;
ssl_certificate  rsa.key;
```

ssl_ciphers

Syntax	`ssl_ciphers` *ciphers*;
Default	`ssl_ciphers HIGH:!aNULL:!MD5;`
Context	http, server

Specifies the enabled ciphers. The ciphers are specified in the format understood by the OpenSSL library, for example:

```
ssl_ciphers ALL:!aNULL:!EXPORT56:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP;
```

The list of ciphers depends on the version of OpenSSL installed. The full list can be viewed using the `openssl ciphers` command.

Attention

The `ssl_ciphers` directive does *not* configure ciphers for TLS 1.3 when using OpenSSL. To tune TLS 1.3 ciphers with OpenSSL, use the *ssl_conf_command* directive, which was added to support advanced SSL configuration.

- In LibreSSL, TLS 1.3 ciphers *can* be configured using `ssl_ciphers`.

- In BoringSSL, TLS 1.3 ciphers cannot be configured at all.

ssl_client_certificate

Syntax	`ssl_client_certificate` *file*;
Default	—
Context	http, server

Specifies a file with trusted CA certificates in the PEM format used to *verify* client certificates and OCSP responses if *ssl_stapling* is enabled.

The list of certificates will be sent to clients. If this is not desired, the *ssl_trusted_certificate* directive can be used.

ssl_conf_command

Syntax	`ssl_conf_command` *name value*;
Default	—
Context	http, server

Sets arbitrary OpenSSL configuration commands.

Important

The directive is supported when using OpenSSL 1.0.2 or higher. To configure TLS 1.3 ciphers with OpenSSL, use the `ciphersuites` command.

Several *ssl_conf_command* directives can be specified on the same level:

```
ssl_conf_command Options PrioritizeChaCha;
ssl_conf_command Ciphersuites TLS_CHACHA20_POLY1305_SHA256;
```

These directives are inherited from the previous configuration level if and only if there are no *ssl_conf_command* directives defined on the current level.

Caution

Configuring OpenSSL directly might result in unexpected behavior.

ssl_crl

Syntax	`ssl_crl` *file*;
Default	—
Context	http, server

Specifies a file with revoked certificates (CRL) in the PEM format used to *verify* client certificates.

ssl_dhparam

Syntax	`ssl_dhparam` *file*;
Default	—
Context	http, server

Specifies a file with DH parameters for DHE ciphers.

By default no parameters are set, and therefore DHE ciphers will not be used.

ssl_early_data

Syntax	`ssl_early_data on | off;`
Default	`ssl_early_data off;`
Context	http, server

Enables or disables TLS 1.3 early data.

Requests sent within early data are subject to replay attacks. To protect against such attacks at the application layer, the *$ssl_early_data* variable should be used.

```
proxy_set_header Early-Data $ssl_early_data;
```

Important

The directive is supported when using OpenSSL 1.1.1 or higher and BoringSSL.

ssl_ecdh_curve

Syntax	`ssl_ecdh_curve` *curve*;
Default	`ssl_ecdh_curve auto;`
Context	http, server

Specifies a curve for ECDHE ciphers.

Important

When using OpenSSL 1.0.2 or higher, it is possible to specify multiple curves, for example:

```
ssl_ecdh_curve prime256v1:secp384r1;
```

The special value `auto` instructs Angie to use a list built into the OpenSSL library when using OpenSSL 1.0.2 or higher, or *prime256v1* with older versions.

Important

When using OpenSSL 1.0.2 or higher, this directive sets the list of curves supported by the server. Thus, in order for ECDSA certificates to work, it is important to include the curves used in the certificates.

ssl_ntls

Added in version 1.2.0.

Syntax	`ssl_ntls on	off;`
Default	`ssl_ntls off;`
Context	http, server

Enables server-side support for NTLS using TongSuo library.

```
listen ... ssl;
ssl_ntls  on;
```

Important

Build Angie using the *--with-ntls* build option and link with NTLS-enabled SSL library

```
./configure --with-openssl=../Tongsuo-8.3.0 \
            --with-openssl-opt=enable-ntls  \
            --with-ntls
```

ssl_ocsp

Syntax	`ssl_ocsp on	off	leaf;`
Default	`ssl_ocsp off;`
Context	http, server

Enables OCSP validation of the client certificate chain. The `leaf` parameter enables validation of the client certificate only.

For the OCSP validation to work, the *ssl_verify_client* directive should be set to `on` or `optional`.

To resolve the OCSP responder hostname, the *resolver* directive should also be specified.

Example:

```
ssl_verify_client on;
ssl_ocsp          on;
resolver          127.0.0.53;
```

ssl_ocsp_cache

Syntax	`ssl_ocsp_cache off` | [shared:*name*:*size*];
Default	`ssl_ocsp_cache off;`
Context	http, server

Sets name and size of the cache that stores client certificates status for OCSP validation. The cache is shared between all worker processes. A cache with the same name can be used in several virtual servers.

The `off` parameter prohibits the use of the cache.

ssl_ocsp_responder

Syntax	`ssl_ocsp_responder` *uri*;
Default	—
Context	http, server

Overrides the URI of the OCSP responder specified in the "Authority Information Access" certificate extension for *validation* of client certificates.

Only `http://` OCSP responders are supported:

```
ssl_ocsp_responder http://ocsp.example.com/;
```

ssl_password_file

Syntax	`ssl_password_file` *file*;
Default	—
Context	http, server

Specifies a file with passphrases for *secret keys* where each passphrase is specified on a separate line. Passphrases are tried in turn when loading the key.

Example:

```
http {
    ssl_password_file /etc/keys/global.pass;
    ...

    server {
        server_name www1.example.com;
        ssl_certificate_key /etc/keys/first.key;
    }

    server {
        server_name www2.example.com;

        # named pipe can also be used instead of a file
        ssl_password_file /etc/keys/fifo;
        ssl_certificate_key /etc/keys/second.key;
    }
}
```

ssl_prefer_server_ciphers

Syntax	`ssl_prefer_server_ciphers on | off;`
Default	`ssl_prefer_server_ciphers off;`
Context	http, server

Specifies that server ciphers should be preferred over client ciphers when using the SSLv3 and TLS protocols.

ssl_protocols

Syntax	`ssl_protocols` [SSLv2] [SSLv3] [TLSv1] [TLSv1.1] [TLSv1.2] [TLSv1.3];
Default	`ssl_protocols TLSv1.2 TLSv1.3;`
Context	http, server

Changed in version 1.2.0: `TLSv1.3` parameter added to default set.

Enables the specified protocols.

Important

The TLSv1.1 and TLSv1.2 parameters work only when OpenSSL 1.0.1 or higher is used.

The TLSv1.3 parameter works only when OpenSSL 1.1.1 or higher is used.

ssl_reject_handshake

Syntax	`ssl_reject_handshake on` | `off`;
Default	`ssl_reject_handshake off;`
Context	http, server

If enabled, SSL handshakes in the *server* block will be rejected.

For example, in the following configuration, SSL handshakes with server names other than *example.com* are rejected:

```
server {
    listen                443 ssl default_server;
    ssl_reject_handshake on;
}

server {
    listen              443 ssl;
    server_name         example.com;
    ssl_certificate     example.com.crt;
    ssl_certificate_key example.com.key;
}
```

ssl_session_cache

Syntax	`ssl_session_cache off` | `none` | [builtin[:*size*]] [shared:*name*:*size*];
Default	`ssl_session_cache none;`
Context	http, server

Sets the types and sizes of caches that store session parameters. A cache can be of any of the following types:

`off`	the use of a session cache is strictly prohibited: Angie explicitly tells a client that sessions may not be reused.
`none`	the use of a session cache is gently disallowed: Angie tells a client that sessions may be reused, but does not actually store session parameters in the cache.
`builtin`	a cache built in OpenSSL; used by one worker process only. The cache size is specified in sessions. If size is not given, it is equal to 20480 sessions. Use of the built-in cache can cause memory fragmentation.
`shared`	a cache shared between all worker processes. The cache size is specified in bytes; one megabyte can store about 4000 sessions. Each shared cache should have an arbitrary name. A cache with the same name can be used in several virtual servers. It is also used to automatically generate, store, and periodically rotate TLS session ticket keys unless configured explicitly using the *ssl_session_ticket_key* directive.

Both cache types can be used simultaneously, for example:

```
ssl_session_cache builtin:1000 shared:SSL:10m;
```

but using only shared cache without the built-in cache should be more efficient.

ssl_session_ticket_key

Syntax	`ssl_session_ticket_key` *file*;
Default	—
Context	http, server

Sets a file with the secret key used to encrypt and decrypt TLS session tickets. The directive is necessary if the same key has to be shared between multiple servers. By default, a randomly generated key is used.

If several keys are specified, only the first key is used to encrypt TLS session tickets. This allows configuring key rotation, for example:

```
ssl_session_ticket_key current.key;
ssl_session_ticket_key previous.key;
```

The file must contain 80 or 48 bytes of random data and can be created using the following command:

```
openssl rand 80 > ticket.key
```

Depending on the file size either AES256 (for 80-byte keys) or AES128 (for 48-byte keys) is used for encryption.

ssl_session_tickets

Syntax	`ssl_session_tickets on` | `off`;
Default	`ssl_session_tickets on;`
Context	http, server

Enables or disables session resumption through TLS session tickets.

ssl_session_timeout

Syntax	`ssl_session_timeout` *time*;
Default	`ssl_session_timeout 5m;`
Context	http, server

Specifies a time during which a client may reuse the session parameters.

ssl_stapling

Syntax	`ssl_stapling on` | `off`;
Default	`ssl_stapling off;`
Context	http, server

Enables or disables stapling of OCSP responses by the server. Example:

```
ssl_stapling on;
resolver 127.0.0.53;
```

For the OCSP stapling to work, the certificate of the server certificate issuer should be known. If the *ssl_certificate* file does not contain intermediate certificates, the certificate of the server certificate issuer should be present in the *ssl_trusted_certificate* file.

Attention

For a resolution of the OCSP responder hostname, the *resolver* directive should also be specified.

ssl_stapling_file

Syntax	`ssl_stapling_file` *file*;
Default	—
Context	http, server

When set, the stapled OCSP response will be taken from the specified file instead of querying the OCSP responder specified in the server certificate.

The file should be in the DER format as produced by the `openssl ocsp` command.

ssl_stapling_responder

Syntax	`ssl_stapling_responder uri`;
Default	—
Context	http, server

Overrides the URI of the OCSP responder specified in the "Authority Information Access". certificate extension.

Only `http://` OCSP responders are supported:

```
ssl_stapling_responder http://ocsp.example.com/;
```

ssl_stapling_verify

Syntax	`ssl_stapling_verify on | off;`
Default	`ssl_stapling_verify off;`
Context	http, server

Enables or disables verification of OCSP responses by the server.

For verification to work, the certificate of the server certificate issuer, the root certificate, and all intermediate certificates should be configured as trusted using the *ssl_trusted_certificate* directive.

ssl_trusted_certificate

Syntax	`ssl_trusted_certificate` *file*;
Default	—
Context	http, server

Specifies a file with trusted CA certificates in the PEM format used to *verify* client certificates and OCSP responses if *ssl_stapling* is enabled.

In contrast to the certificate set by *ssl_client_certificate*, the list of these certificates will not be sent to clients.

ssl_verify_client

Syntax	`ssl_verify_client on | off | optional | optional_no_ca;`
Default	`ssl_verify_client off;`
Context	http, server

Enables verification of client certificates. The verification result is stored in the *$ssl_client_verify* variable.

`optional`	requests the client certificate and verifies it if the certificate is present.
`optional_no_ca`	requests the client certificate but does not require it to be signed by a trusted CA certificate. This is intended for the use in cases when a service that is external to Angie performs the actual certificate verification.

ssl_verify_depth

Syntax	`ssl_verify_depth` *number*;
Default	`ssl_verify_depth 1;`
Context	http, server

Sets the verification depth in the client certificates chain.

Error Processing

The `http_ssl` module supports several non-standard error codes that can be used for redirects using the *error_page* directive:

495	an error has occurred during the client certificate verification;
496	a client has not presented the required certificate;
497	a regular request has been sent to the HTTPS port.

The redirection happens after the request is fully parsed and the variables, such as *$request_uri*, *$uri*, *$args* and others, are available.

Built-in Variables

The *http_ssl* module supports built-in variables:

`$ssl_alpn_protocol`

returns the protocol selected by ALPN during the SSL handshake, or an empty string otherwise.

`$ssl_cipher`

returns the name of the cipher used for an established SSL connection.

`$ssl_ciphers`

returns the list of ciphers supported by the client. Known ciphers are listed by names, unknown are shown in hexadecimal, for example:

AES128-SHA:AES256-SHA:0x00ff

Important

The variable is fully supported only when using OpenSSL version 1.0.2 or higher. With older versions, the variable is available only for new sessions and lists only known ciphers.

`$ssl_client_escaped_cert`

returns the client certificate in the PEM format (urlencoded) for an established SSL connection;

`$ssl_client_fingerprint`

returns the SHA1 fingerprint of the client certificate for an established SSL connection;

`$ssl_client_i_dn`

returns the "issuer DN" string of the client certificate for an established SSL connection according to RFC 2253;

`$ssl_client_i_dn_legacy`

returns the "issuer DN" string of the client certificate for an established SSL connection.

`$ssl_client_raw_cert`

returns the client certificate in the PEM format for an established SSL connection.

`$ssl_client_s_dn`

returns the "subject DN" string of the client certificate for an established SSL connection according to RFC 2253.

`$ssl_client_s_dn_legacy`

returns the "subject DN" string of the client certificate for an established SSL connection.

`$ssl_client_serial`

returns the serial number of the client certificate for an established SSL connection.

`$ssl_client_v_end`

returns the end date of the client certificate.

`$ssl_client_v_remain`

returns the number of days until the client certificate expires.

`$ssl_client_v_start`

returns the start date of the client certificate.

`$ssl_client_verify`

returns the result of client certificate verification: `SUCCESS`, `FAILED:reason`, and `NONE` if a certificate was not present.

`$ssl_curve`

returns the negotiated curve used for SSL handshake key exchange process. Known curves are listed by names, unknown are shown in hexadecimal, for example:

prime256v1

Important

The variable is supported only when using OpenSSL version 3.0 or higher. With older versions, the variable value will be an empty string.

`$ssl_curves`

returns the list of curves supported by the client. Known curves are listed by names, unknown are shown in hexadecimal, for example:

0x001d:prime256v1:secp521r1:secp384r1

Important

The variable is supported only when using OpenSSL version 1.0.2 or higher. With older versions, the variable value will be an empty string.

The variable is available only for new sessions.

`$ssl_early_data`

returns "1" if TLS 1.3 *early data* is used and the handshake is not complete, otherwise "".

`$ssl_protocol`

returns the protocol of an established SSL connection.

`$ssl_server_cert_type`

takes the values `RSA`, `DSA`, `ECDSA`, `ED448`, `ED25519`, `SM2`, `RSA-PSS`, or `unknown` depending on the type of server certificate and key.

`$ssl_server_name`

returns the server name requested through SNI;

`$ssl_session_id`

returns the session identifier of an established SSL connection.

`$ssl_session_reused`

returns `r` if an SSL session was reused, or "." otherwise.

Stub Status

The module provides access to basic status information.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-http_stub_status_module` build option.

In packages and images from our repos, the module is included in the build.

Configuration Example

```
location = /basic_status {
    stub_status;
}
```

This configuration creates a simple web page with basic status data which may look like as follows:

```
Active connections: 291
server accepts handled requests
 16630948 16630948 31070465
Reading: 6 Writing: 179 Waiting: 106
```

Directives

stub_status

Syntax	`stub_status;`
Default	—
Context	server, location

The basic status information will be accessible from the surrounding location.

Data

The following status information is provided:

`Active connections`

The current number of active client connections including Waiting connections.

`accepts`

The total number of accepted client connections.

`handled`

The total number of handled connections. Generally, the parameter value is the same as accepts unless some resource limits have been reached (for example, the *worker_connections* limit).

`requests`

The total number of client requests.

`Reading`

The current number of connections where Angie is reading the request header.

`Writing`

The current number of connections where Angie is writing the response back to the client.

`Waiting`

The current number of idle client connections waiting for a request.

Built-in Variables

`$connections_active`

Same as the *Active connections* value.

`$connections_reading`

Same as the *Reading* value.

`$connections_writing`

Same as the *Writing* value.

`$connections_waiting`

Same as the *Waiting* value.

Sub

The module is a filter that modifies a response by replacing one specified string with another.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-http_sub_module` build option.

In packages and images from our repos, the module is included in the build.

Configuration Example

```
location / {
    sub_filter '<a href="http://127.0.0.1:8080/'  '<a href="https://$host/';
    sub_filter '<img src="http://127.0.0.1:8080/' '<img src="https://$host/';
    sub_filter_once on;
}
```

Directives

sub_filter

Syntax	`sub_filter` *string replacement*;
Default	—
Context	http, server, location

Sets a string to replace and a replacement string. The string to replace is matched ignoring the case. The string to replace and replacement string can contain variables. Several `sub_filter` directives can be specified on the same configuration level. These directives are inherited from the previous configuration level if and only if there are no `sub_filter` directives defined on the current level.

sub_filter_last_modified

Syntax	`sub_filter_last_modified on | off;`
Default	`sub_filter_last_modified off;`
Context	http, server, location

Allows preserving the "Last-Modified" header field from the original response during replacement to facilitate response caching.

By default, the header field is removed as contents of the response are modified during processing.

sub_filter_once

Syntax	`sub_filter_once on | off;`
Default	`sub_filter_once on;`
Context	http, server, location

Indicates whether to look for each string to replace once or repeatedly.

sub_filter_types

Syntax	`sub_filter_types` mime-*type* ...;
Default	`sub_filter_types text/html;`
Context	http, server, location

Enables string replacement in responses with the specified MIME types in addition to `text/html`. The special value "*" matches any MIME type.

Upstream

The module is used to define groups of servers that can be referenced by the *proxy_pass*, *fastcgi_pass*, *uwsgi_pass*, *scgi_pass*, *memcached_pass* and *grpc_pass* directives.

Configuration Example

```
upstream backend {
    zone backend 1m;
    server backend1.example.com       weight=5;
    server backend2.example.com:8080;
    server backend3.example.com       service=_example._tcp resolve;
    server unix:/tmp/backend3;

    server backup1.example.com:8080   backup;
    server backup2.example.com:8080   backup;
}

resolver 127.0.0.53 status_zone=resolver;

server {
    location / {
        proxy_pass http://backend;
    }
}
```

Directives

backup_switch (PRO)

Added in version 1.9.0: PRO

Syntax	`backup_switch permanent`[=`time];`
Default	—
Context	upstream

The directive enables active backups for the `upstream` where it occurs. Once a request fails to select a server in the primary group and resorts to the backup group, that group will become *active* if the directive is defined. Subsequent requests are handled by first looking for servers in the active group.

When `permanent` is defined without a *time* value, the group remains active once selected, and no automatic reevaluation occurs. If the *time* limit is set, the active status times out after the specified *interval*. and the balancer reevaluates the primary group, reverting to it if the servers are healthy.

Example:

```
upstream my_backend {
    server primary1.example.com;
    server primary2.example.com;

    server backup1.example.com backup;
    server backup2.example.com backup;

    backup_switch permanent=2m;
}
```

If the balancer fails over from the primary servers to the backup group, all subsequent requests are served by that backup group for 2 minutes. Once 2 minutes elapse, the balancer reevaluates the primary servers

and makes them active again if they're found to be healthy.

bind_conn (PRO)

Syntax	`bind_conn` *value*;
Default	—
Context	upstream

Enables binding the server connection to the client when the *value*, which is set as a string of variables, becomes anything other than `""` and `"0"`.

Attention

The `bind_conn` directive must be used after all directives that set the load balancing method; otherwise, it won't work. If *sticky* is also used, `bind_conn` should appear after `sticky`.

Attention

When using the directive, configure the *http_proxy* module to allow keepalive connections, for example:

```
proxy_http_version 1.1;
proxy_set_header Connection "";
```

A typical use case for the directive is proxying NTLM-authenticated connections, where the client should be bound to the server when the negotiation starts:

```
map $http_authorization   $ntlm {
    ~*^N(?:TLM|egotiate)  1;
}

upstream ntlm_backend {
    server 127.0.0.1:8080;
    bind_conn $ntlm;
}

server {
    # ...
    location / {
        proxy_pass http://ntlm_backend;
        proxy_http_version 1.1;
        proxy_set_header Connection "";
        # ...
    }
}
```

feedback (PRO)

Added in version 1.6.0: PRO

Syntax	`feedback` *variable* [`inverse`] [`factor=`*number*] [`account=`*condition_variable*] [`last_byte`];
Default	—
Context	upstream

Enables a feedback-based load balancing mechanism for the `upstream`. It adjusts the load balancing decisions dynamically, multiplying each peer's weight by its average feedback value that is affected by the value of a *variable* over time and is subject to an optional condition.

The following parameters are accepted:

`variable`	The variable from which the feedback value is taken. It should represent a performance or health metric, and is intended to be supplied by the peer in header fields or otherwise. The value is assessed at each response from the peer and factored into the rolling average according to `inverse` and `factor` settings.
`inverse`	If set, the feedback value is interpreted inversely, meaning lower values indicate better performance.
`factor`	The factor by which the feedback value is weighted when calculating the average. Valid values are integers between 0 and 99. By default — `90`. The average feedback is calculated using the exponential moving average formula. The larger is the factor, the less is the average affected by new values; if the factor is set to `90`, the result has 90% of the previous value and only 10% of the new value.
`account`	Specifies a condition variable that controls which responses should be included in the calculation. The average is updated with the feedback value only if the condition variable for the response isn't `""` or `"0"`. **Note** By default, responses from *probes* aren't included in the calculation; combining the *$upstream_probe* variable with `account` allows to include these responses or even exclude everything else.
`last_byte`	Allows processing feedback from the upstream server after the full response has been received, instead of just after the header.

Example:

```
upstream backend {

    zone backend 1m;

    feedback $feedback_value factor=80 account=$condition_value;

    server backend1.example.com;
    server backend2.example.com;
}

map $upstream_http_custom_score $feedback_value {
    "high"                      100;
    "medium"                    75;
    "low"                       50;
    default                     10;
}

map $upstream_probe $condition_value {
```

```
    "high_priority" "1";
    "low_priority"  "0";
    default         "1";
}
```

This categorizes server responses into different feedback levels based on specific scores obtained from response header fields, and also adds a condition mapped from *$upstream_probe* to account only for the responses from the `high_priority` probe or responses to regular client requests.

hash

Syntax	`hash` *key* [`consistent`];
Default	—
Context	upstream

Specifies a load balancing method for a server group where the client-server mapping is based on the hashed key value. The key can contain text, variables, and their combinations. Note that adding or removing a server from the group may result in remapping most of the keys to different servers. The method is compatible with the Cache::Memcached Perl library.

If the `consistent` parameter is specified, the ketama consistent hashing method will be used instead. The method ensures that only a few keys will be remapped to different servers when a server is added to or removed from the group. This helps to achieve a higher cache hit ratio for caching servers. The method is compatible with the Cache::Memcached::Fast Perl library with the ketama_points parameter set to 160.

ip_hash

Syntax	`ip_hash`;
Default	—
Context	upstream

Specifies that a group should use a load balancing method where requests are distributed between servers based on client IP addresses. The first three octets of the client IPv4 address, or the entire IPv6 address, are used as a hashing key. The method ensures that requests from the same client will always be passed to the same server except when this server is unavailable. In the latter case client requests will be passed to another server. Most probably, it will always be the same server as well.

If one of the servers needs to be temporarily removed, it should be marked with the `down` parameter in order to preserve the current hashing of client IP addresses.

```
upstream backend {
    ip_hash;

    server backend1.example.com;
    server backend2.example.com;
    server backend3.example.com down;
    server backend4.example.com;
}
```

keepalive

Syntax	`keepalive` *connections*;
Default	—
Context	upstream

Activates the cache for connections to upstream servers.

The `connections` parameter sets the maximum number of idle keepalive connections to upstream servers that are preserved in the cache of each worker process. When this number is exceeded, the least recently used connections are closed.

Note

It should be particularly noted that the *keepalive* directive does not limit the total number of connections to upstream servers that an Angie worker process can open. The connections parameter should be set to a number small enough to let upstream servers process new incoming connections as well.

Attention

The `keepalive` directive must be used after all directives that set the load balancing method; otherwise, it won't work.

Example configuration of memcached upstream with keepalive connections:

```
upstream memcached_backend {
    server 127.0.0.1:11211;
    server 10.0.0.2:11211;

    keepalive 32;
}

server {
    #...

    location /memcached/ {
        set $memcached_key $uri;
        memcached_pass memcached_backend;
    }

}
```

For HTTP, the *proxy_http_version* directive should be set to "1.1" and the "Connection" header field should be cleared:

```
upstream http_backend {
    server 127.0.0.1:8080;

    keepalive 16;
}

server {
    #...

    location /http/ {
        proxy_pass http://http_backend;
        proxy_http_version 1.1;
        proxy_set_header Connection "";
    #    ...
    }
}
```

Note

Alternatively, HTTP/1.0 persistent connections can be used by passing the "Connection: Keep-Alive" header field to an upstream server, though this method is not recommended.

For FastCGI servers, it is required to set *fastcgi_keep_conn* for keepalive connections to work:

```
upstream fastcgi_backend {
    server 127.0.0.1:9000;

    keepalive 8;
}

server {
    #...

    location /fastcgi/ {
        fastcgi_pass fastcgi_backend;
        fastcgi_keep_conn on;
    #    ...
    }
}
```

Note

SCGI and uwsgi protocols do not have a notion of keepalive connections.

keepalive_requests

Syntax	`keepalive_requests` *number*;
Default	`keepalive_requests 1000;`
Context	upstream

Sets the maximum number of requests that can be served through one keepalive connection. After the maximum number of requests is made, the connection is closed.

Closing connections periodically is necessary to free per-connection memory allocations. Therefore, using too high maximum number of requests could result in excessive memory usage and not recommended.

keepalive_time

Syntax	`keepalive_time` *time*;
Default	`keepalive_time 1h;`
Context	upstream

Limits the maximum time during which requests can be processed through one keepalive connection. After this time is reached, the connection is closed following the subsequent request processing.

keepalive_timeout

Syntax	`keepalive_timeout` *time*;
Default	`keepalive_timeout 60s;`
Context	upstream

Sets a timeout during which an idle keepalive connection to an upstream server will stay open.

least_conn

Syntax	`least_conn`;
Default	—
Context	upstream

Specifies that a group should use a load balancing method where a request is passed to the server with the least number of active connections, taking into account weights of servers. If there are several such servers, they are tried in turn using a weighted round-robin balancing method.

least_time (PRO)

Syntax	`least_time header` | `last_byte` [`factor=`*number*] [`account=`*condition_variable*];
Default	—
Context	upstream

Specifies that the group should use a load balancing method where an active server's chance of receiving the request is inversely proportional to its average response time; the less it is, the more requests a server gets.

`header`	The directive accounts for response headers only.
`last_byte`	The directive uses the average time to receive the entire response.

Added in version 1.7.0: PRO

`factor`	Serves the same purpose as *response_time_factor (PRO)* and overrides it if set.
`account`	Specifies a condition variable that controls which responses should be included in the calculation. The average is updated only if the condition variable for the response isn't `""` or `"0"`.

Note

By default, responses from *probes* aren't included in the calculation; combining the *$upstream_probe* variable with `account` allows to include these responses or even exclude everything else.

The respective moving averages, adjusted for `factor` and `account`, are also presented as `header_time` and `response_time` in the server's `health` object among the *upstream metrics* in the API.

queue (PRO)

Added in version 1.4.0: PRO

Syntax	`queue` *number* [`timeout=`*time*];
Default	—
Context	upstream

If it is not possible to assign a proxied server to a request on the first attempt (for example, during a brief service interruption or when there is a surge in load reaching the *max_conns* limit), the request is not rejected; instead, Angie attempts to enqueue it for processing.

The number in the directive sets the maximum number of requests in the queue for a *worker process*. If the queue is full, a `502 (Bad Gateway)` error is returned to the client.

Note

The logic of the *proxy_next_upstream* directive also applies to queued requests. Specifically, if a server was selected for a request but it cannot be handed over to it, the request may be returned to the queue.

If a server is not selected to process a queued request within the *time* set by `timeout` (default is 60 seconds), a `502 (Bad Gateway)` error is returned to the client. Requests from clients that prematurely close the connection are also removed from the queue; there are counters for requests passing through the queue in the *API*.

Attention

The `queue` directive must be used after all directives that set the load balancing method; otherwise, it won't work.

random

Syntax	`random` [`two`];
Default	—
Context	upstream

Specifies that a group should use a load balancing method where a request is passed to a randomly selected server, taking into account weights of servers.

The optional `two` parameter instructs Angie to randomly select two servers and then choose a server using the specified method. The default method is *least_conn* which passes a request to a server with the least number of active connections.

response_time_factor (PRO)

Syntax	`response_time_factor` *number*;
Default	`response_time_factor 90;`
Context	upstream

If the *least_time (PRO)* load balancing method is used, sets the smoothing factor for the **previous** value when average response time is calculated using the exponential moving average formula.

The larger is the *number*, the less is the average affected by new values; if the *number* is set to `90`, the result has 90% of the previous value and only 10% of the new value. The allowed range is 0 to 99, inclusive.

The respective moving averages are presented as `header_time` (headers only) and `response_time` (entire responses) in the server's `health` object among the *upstream metrics* in the API.

Note

The calculation accounts for successful reponses only; what is considered an unsuccessful response is defined by the *proxy_next_upstream*, *fastcgi_next_upstream*, *uwsgi_next_upstream*, *scgi_next_upstream*, *memcached_next_upstream*, and *grpc_next_upstream* directives. Besides, `header_time` is updated only if all headers are received and processed, and `response_time` is updated only if the entire reponse is received.

server

Syntax	`server` *address* [*parameters*];
Default	—
Context	upstream

Defines the address and other parameters of a server. The address can be specified as a domain name or IP address, with an optional port, or as a UNIX domain socket path specified after the `unix:` prefix. If a port is not specified, the port 80 is used. A domain name that resolves to several IP addresses defines multiple servers at once.

The following parameters can be defined:

`weight=`*number*	sets the weight of the server; by default, 1.
`max_conns=`*number*	limits the maximum number of simultaneous active connections to the proxied server. Default value is `0`, meaning there is no limit. If the server group does not reside in the *shared memory*, the limitation works per each worker process.

Note

If idle *keepalive* connections, multiple *workers*, and the *shared memory* are enabled, the total number of active and idle connections to the proxied server may exceed the *max_conns* value.

`max_fails=`*number* — sets the number of unsuccessful attempts to communicate with the server that should happen in the duration set by *fail_timeout* to consider the server unavailable; it is then retried after the same duration.

What is considered an unsuccessful attempt is defined by the *proxy_next_upstream*, *fastcgi_next_upstream*, *uwsgi_next_upstream*, *scgi_next_upstream*, *memcached_next_upstream*, and *grpc_next_upstream* directives.

When `max_fails` is reached, the peer is also considered unhealthy by the *upstream_probe (PRO)* probes; it won't receive client requests until the probes consider it healthy again.

Note

If a `server` in an upstream resolves into multiple peers, its `max_fails` setting applies to each peer individually.

If an upstream contains only one peer after all its `server` directives are resolved, the `max_fails` setting has no effect and will be ignored.

`max_fails=1`	the default number of unsuccessful attempts
`max_fails=0`	disables the accounting of attempts

`fail_timeout=`*time* — sets the period of time during which a number of unsuccessful attempts to communicate with the server (*max_fails*) should happen to consider the server unavailable. The server then becomes unavailable for the same amount of time before it is retried.

By default, this is set to 10 seconds.

Note

If a `server` in an upstream resolves into multiple peers, its `fail_timeout` setting applies to each peer individually.

If an upstream contains only one peer after all its `server` directives are resolved, the `fail_timeout` setting has no effect and will be ignored.

`backup`	marks the server as a backup server. It will be passed requests when the primary servers are unavailable. If *backup_switch (PRO)* is configured, its active backup logic is also applied.
`down`	marks the server as permanently unavailable.
`drain`	sets the server to draining; this means it receives only requests from the sessions that were bound earlier with *sticky*. Otherwise it behaves similarly to `down`.

Caution

The parameter `backup` cannot be used along with the *hash*, *ip_hash*, and *random* load balancing methods.

The `down` and `drain` options are mutually exclusive.

Added in version 1.1.0.

`resolve`	enables monitoring changes to the list of IP addresses that corresponds to a domain name, updating it without a configuration reload. The group should be stored in a *shared memory zone*; also, you need to define a *resolver*.
`service=`*name*	enables resolving DNS SRV records and sets the service name. For this parameter to work, specify the *resolve* server parameter, providing a hostname without a port number. If there are no dots in the service name, the name is formed according to the RFC standard: the service name is prefixed with `_`, then `_tcp` is added after a dot. Thus, the service name `http` will result in `_http._tcp`. Angie resolves the SRV records by combining the normalized service name and the hostname and obtaining the list of servers for the combination via DNS, along with their priorities and weights. • Top-priority SRV records (ones that share the minimum priority value) resolve into primary servers, and other records become backup servers. If `backup` is set with `server`, top-priority SRV records resolve into backup servers, and other records are ignored. • Weight influences the selection of servers by the assigned capacity: higher weights receive more requests. If set by both the `server` directive and the SRV record, the weight set by `server` is used.

This example will look up the `_http._tcp.backend.example.com` record:

```
server backend.example.com service=http resolve;
```

Added in version 1.2.0: Angie

Added in version 1.1.0-P1: Angie PRO

`sid=`*id*	sets the server ID within the group. If the parameter is omitted, the ID is set to the hexadecimal MD5 hash value of either the IP address and port or the UNIX domain socket path.

Added in version 1.4.0.

`slow_start=`*time*	sets the *time* to recover the `weight` for a server that goes back online, if load balancing uses the *round-robin* or *least_conn* method. If the value is set and the server is again considered available and healthy as defined by *max_fails* and *upstream_probe (PRO)*, the server will steadily recover its designated weight within the allocated timeframe. If the value isn't set, the server in a similar situation will recover its designated weight immediately.

Note

If there's only one `server` in an upstream, `slow_start` has no effect and will be ignored.

state (PRO)

Added in version 1.2.0: PRO

Syntax	`state` *file*;
Default	—
Context	upstream

Specifies the *file* where the upstream's server list is persisted. When installing from our packages, a designated `/var/lib/angie/state/` (`/var/db/angie/state/` on FreeBSD) directory with appropriate permissions is created to store these files, so you will only need to add the file's basename in the configuration:

```
upstream backend {

    zone backend 1m;
    state /var/lib/angie/state/<FILE NAME>;
}
```

The format of this server list is similar to `server`. The contents of the file change whenever there is any modification to servers in the */config/http/upstreams/* section via the configuration API. The file is read at Angie start or configuration reload.

Caution

For the `state` directive to be used in an `upstream` block, the block should have no `server` directives; instead, it must have a shared memory zone (*zone*).

sticky

Added in version 1.2.0: Angie

Added in version 1.1.0-P1: Angie PRO

Syntax	`sticky` cookie *name* [*attr*=*value*]...; `sticky` route *$variable*...; `sticky` learn `zone`=*zone* `create`=*$create_var1*... `lookup`=*$lookup_var1*... [`header`] [`timeout`=*time*]; `sticky` learn `zone`=*zone* `lookup`=*$lookup_var1*... `remote_action`=*uri* `remote_result`=*$remote_var* [`timeout`=*time*];
Default	—
Context	upstream

Configures the binding of client sessions to proxied servers in the mode specified by the first parameter; to drain requests from servers that have `sticky` defined, use the `drain` option in the *server* block.

Attention

The `sticky` directive must be used after all directives that set the load balancing method; otherwise, it won't work. If *bind_conn (PRO)* is also used, `bind_conn` should appear after `sticky`.

`cookie` mode

This mode uses cookies to maintain session persistence. It is more suitable for situations where cookies are already used for session management.

Here, a client's request, not yet bound to any server, is sent to a server chosen according to the configured balancing method. Also, Angie sets a cookie with a unique value identifying the server.

The cookie's name (`name`) is set by the `sticky` directive, and the value (`value`) corresponds to the *sid* parameter of the *server* directive. Note that the parameter is additionally hashed if the *sticky_secret* directive is set.

Subsequent client requests that contain this cookie are forwarded to the server identified by the cookie's value, which is the server with the specified *sid*. If selecting a server fails or the chosen server can't handle the request, another server is selected according to the configured balancing method.

The directive allows assigning attributes to the cookie; the only attribute set by default is `path=/`. Attribute values are specified as strings with variables. To remove an attribute, set an empty value for it: `attr=`. Thus, `sticky cookie path=` creates a cookie without `path`.

Here, Angie creates a cookie named `srv_id` with a one-hour lifespan and a variable-specified domain:

```
upstream backend {
    server backend1.example.com:8080;
    server backend2.example.com:8080;

    sticky cookie srv_id domain=$my_domain max-age=3600;
}
```

`route` mode

This mode uses predefined route identifiers that can be embedded in URLs, cookies, or other request properties. It is less flexible because it relies on predefined values but can suit better if such identifiers are already in place.

Here, when a proxied server receives a request, it can assign a route to the client and return its identifier in a way that both the client and the server are aware of. The value of the *sid* parameter of the *server* directive must be used as the route identifier. Note that the parameter is additionally hashed if the *sticky_secret* directive is set.

Subsequent requests from clients that wish to use this route must contain the identifier issued by the server in a way that ensures it ends up in Angie variables, for example, in *cookie* or *request arguments*.

The directive lists the specific variables used for routing. To select the server to which the incoming request is forwarded, the first non-empty variable is used; it is then compared with the *sid* parameter of the *server* directive. If selecting a server fails or the chosen server can't handle the request, another server is selected according to the configured balancing method.

Here, Angie looks for the route identifier in the `route` cookie, and then in the `route` request argument:

```
upstream backend {
    server backend1.example.com:8080 "sid=server 1";
    server backend2.example.com:8080 "sid=server 2";

    sticky route $cookie_route $arg_route;
}
```

`learn` mode (PRO 1.4.0+)

This mode uses a dynamically generated key to associate a client with a particular proxied server; it's more flexible because it assigns servers on the go, stores sessions in a shared memory zone, and supports different ways of passing session identifiers.

Here, a session is created based on the response from the proxied server. The `create` and `lookup` parameters list variables indicating how new sessions are created and existing sessions are looked up. Both parameters can occur multiple times.

The session identifier is the value of the first non-empty variable specified with `create`; for example, this could be a *cookie from the proxied server*.

Sessions are stored in a shared memory zone; its name and size are set by the `zone` parameter. If a session has been inactive for the time set by `timeout`, it is deleted. The default is 10 minutes.

Subsequent requests from clients that wish to use the session must contain its identifier, ensuring that it ends up in a non-empty variable specified with `lookup`; its value will then be matched against sessions in shared memory. If selecting a server fails or the chosen server can't handle the request, another server is selected according to the configured balancing method.

The `header` parameter allows creating a session immediately after receiving headers from the proxied server. Without it, a session is created only after processing the request.

In the example, Angie creates a session, setting a cookie named `examplecookie` in the response:

```
upstream backend {
    server backend1.example.com:8080;
    server backend2.example.com:8080;

    sticky learn
        lookup=$cookie_examplecookie
        zone=client_sessions:1m;
}
```

`learn` mode with `remote_action` (PRO 1.8.0+)

The `remote_action` and `remote_result` parameters enable dynamically assigning and managing session IDs via remote session storage. Here, the shared memory acts as a local cache, while the remote storage is the authoritative source. Thus, the `create` parameter is incompatible with `remote_action` because session IDs need to be created remotely. If a session has been inactive for the time set by `timeout`, it is deleted. The `remote_action` setting doesn't affect the timeout. The default is 10 minutes.

The initial session ID always comes from `lookup`; if it can be found in the local shared memory, Angie proceeds to select the appropriate peer.

If this session ID isn't found locally, Angie sends a synchronous subrequest to remote storage. The `remote_action` parameter sets the URI of the remote storage, which should handle session lookup and creation as follows:

- It accepts the session ID from `lookup` and the locally suggested server ID to be associated with this session via custom header fields or in some other way. On Angie's side, two special variables are provided for this purpose: *$sticky_sessid* and *$sticky_sid*, respectively. The `sticky_sid` value comes from the `sid=` setting in the *upstream* block of the `server` directive, if it's set; otherwise, it is an MD5 hash of the server's name.
- A 200 response from the remote storage indicates it has accepted the session and saved it with the suggested values for later retrieval.
- A 409 response from the remote storage indicates that this session ID is already populated. In this case, the response should suggest an alternative session ID in the `X-Sticky-Sid` header field. Angie saves this ID in the variable set by the `remote_result` parameter.

In this example, Angie creates a session, uses the `$cookie_bar` variable for the initial session ID, and stores alternative session IDs reported by the remote storage in `$upstream_http_x_sticky_sid`:

```
http {

    upstream u1 {

        server srv1;
        server srv2;

        sticky learn zone=sz:1m
            lookup=$cookie_bar
            remote_action=/remote_session
            remote_result=$upstream_http_x_sticky_sid;

        zone z 1m;
```

```
    }

    server {

        listen localhost;

        location / {

            proxy_pass http://u1/;
        }

        location /remote_session {

            internal;
            proxy_set_header X-Sticky-Sessid $sticky_sessid;
            proxy_set_header X-Sticky-Sid $sticky_sid;
            proxy_set_header X-Sticky-Last $msec;
            proxy_pass http://remote;
        }
    }
}
```

sticky_secret

Added in version 1.2.0: Angie

Added in version 1.1.0-P1: Angie PRO

Syntax	`sticky_secret` *string*;
Default	—
Context	upstream

Adds the *string* as the salt value to the MD5 hashing function for the *sticky* directive in `cookie` and `route` modes. The *string* may contain variables, for example, *$remote_addr*:

```
upstream backend {
    server backend1.example.com:8080;
    server backend2.example.com:8080;

    sticky cookie cookie_name;
    sticky_secret my_secret.$remote_addr;
}
```

Salt is appended to the value being hashed; to verify the hashing mechanism independently:

```
$ echo -n "<VALUE><SALT>" | md5sum
```

sticky_strict

Added in version 1.2.0: Angie

Added in version 1.1.0-P1: Angie PRO

Syntax	`sticky_strict on` | `off`;
Default	`sticky_strict off;`
Context	upstream

When enabled, causes Angie to return an HTTP 502 error to the client if the desired server is unavailable, rather than using any other available server as it would when no servers in the upstream are available.

upstream

Syntax	`upstream` *name* { ... }
Default	—
Context	http

Defines a group of servers. Servers can listen on different ports. In addition, servers listening on TCP and UNIX domain sockets can be mixed.

Example:

```
upstream backend {
    server backend1.example.com weight=5;
    server 127.0.0.1:8080       max_fails=3 fail_timeout=30s;
    server unix:/tmp/backend3;

    server backup1.example.com  backup;
}
```

By default, requests are distributed between the servers using a weighted round-robin balancing method. In the above example, each 7 requests will be distributed as follows: 5 requests go to backend1.example.com and one request to each of the second and third servers.

If an error occurs during communication with a server, the request will be passed to the next server, and so on until all of the functioning servers will be tried. If a successful response could not be obtained from any of the servers, the client will receive the result of the communication with the last server.

zone

Syntax	`zone` *name* [*size*];
Default	—
Context	upstream

Defines the name and size of the shared memory zone that keeps the group's configuration and run-time state that are shared between worker processes. Several groups may share the same zone. In this case, it is enough to specify the size only once.

Built-in Variables

The `http_upstream` module supports the following built-in variables:

`$sticky_sessid`

Used with `remote_action` in *sticky*; stores the initial session ID taken from `lookup`.

`$sticky_sid`

Used with `remote_action` in *sticky*; stores the server ID tentatively associated with the session.

`$upstream_addr`

stores the IP address and port, or the path to the UNIX domain socket of the upstream server. If several servers were contacted during request processing, their addresses are separated by commas, e.g.:

192.168.1.1:80, 192.168.1.2:80, unix:/tmp/sock

If an internal redirect from one server group to another happens, initiated by "X-Accel-Redirect" or *error_page*, then the server addresses from different groups are separated by colons, e.g.:

192.168.1.1:80, 192.168.1.2:80, unix:/tmp/sock : 192.168.10.1:80, 192.168.10.2:80

If a server cannot be selected, the variable keeps the *name* of the *server group*.

`$upstream_bytes_received`

number of bytes received from an upstream server. Values from several connections are separated by commas and colons like addresses in the *$upstream_addr* variable.

`$upstream_bytes_sent`

number of bytes sent to an upstream server. Values from several connections are separated by commas and colons like addresses in the *$upstream_addr* variable.

`$upstream_cache_status`

keeps the status of accessing a response cache. The status can be either `MISS`, `BYPASS`, `EXPIRED`, `STALE`, `UPDATING`, `REVALIDATED` or `HIT`:

- `MISS`: The response isn't found in the cache, and the request is forwarded to the upstream server.
- `BYPASS`: The cache is bypassed, and the request is directly forwarded to the upstream server.
- `EXPIRED`: The cached response is stale, and a new request for the updated content is sent to the upstream server.
- `STALE`: The cached response is stale, but will be served to the clients until an update has been eventually fetched from the upstream server.
- `UPDATING`: The cached response is stale, but will be served to the clients until the currently ongoing update from the upstream server has been finished.
- `REVALIDATED`: The cached response is stale, but is successfully revalidated and doesn't need an update from the upstream server.
- `HIT`: The response was served from the cache.

If the cache was bypassed entirely without accessing it, the variable isn't set.

`$upstream_connect_time`

keeps time spent on establishing a connection with the upstream server; the time is kept in seconds with millisecond resolution. In case of SSL, includes time spent on handshake. Times of several connections are separated by commas and colons like addresses in the *$upstream_addr* variable.

`$upstream_cookie_<name>`

cookie with the specified *name* sent by the upstream server in the "Set-Cookie" response header field. Only the cookies from the response of the last server are saved.

`$upstream_header_time`

stores time spent on receiving the response header from the upstream server; the time is kept in seconds with millisecond resolution. Times of several responses are separated by commas and colons like addresses in the *$upstream_addr* variable.

`$upstream_http_<name>`

stores server response header fields. For example, the "Server" response header field is available through the *$upstream_http_server* variable. The rules of converting header field names to variable names are the same as for the variables that start with the "$http_" prefix. Only the header fields from the response of the last server are saved.

`$upstream_queue_time`

stores time the request spent in the *queue* before a server was selected; the time is kept in seconds with millisecond resolution. Times of several selection attempts are separated by commas and colons, like addresses in the *$upstream_addr* variable.

`$upstream_response_length`

keeps the length of the response obtained from the upstream server; the length is kept in bytes. Lengths of several responses are separated by commas and colons like addresses in the *$upstream_addr* variable.

`$upstream_response_time`

keeps time spent on receiving the response from the upstream server; the time is kept in seconds with millisecond resolution. Times of several responses are separated by commas and colons like addresses in the *$upstream_addr* variable.

`$upstream_status`

keeps status code of the response obtained from the upstream server. Status codes of several responses are separated by commas and colons like addresses in the *$upstream_addr* variable. If a server cannot be selected, the variable keeps the 502 (Bad Gateway) status code.

`$upstream_sticky_status`

Status of sticky requests.

`""`	Request sent to upstream without sticky enabled.
`NEW`	Request without sticky information.
`HIT`	Request with sticky information routed to the desired backend.
`MISS`	Request with sticky information routed to the backend selected by the load balancing algorithm.

Values from multiple connections are separated by commas and colons, similar to addresses in the *$upstream_addr* variable.

`$upstream_trailer_<name>`

stores fields from the end of the response obtained from the upstream server.

Upstream Probe

The module implements active health probes for *http_upstream*.

Configuration Example

```
server {
    listen ...;

    location /backend {
        ...
        proxy_pass http://backend;

        upstream_probe backend_probe
            uri=/probe
            port=10004
            interval=5s
            test=$good
            essential
            fails=3
            passes=3
            max_body=10m
            mode=idle;
    }
}
```

Directives

upstream_probe (PRO)

Added in version 1.2.0: PRO

Syntax	`upstream_probe` *name* [`uri=`*address*] [`port=`*number*] [`interval=`*time*] [`method=`*method*] [`test=`*condition*] [`essential` [`persistent`]] [`fails=`*number*] [`passes=`*number*] [`max_body=`*number*] [`mode=always` | `idle` | `onfail`];
Default	—
Context	location

Defines an active health probe for peers within the *upstream* groups that are specified for *proxy_pass*, *uwsgi_pass*, and so on in the same `location` context with the `upstream_probe` directive. Subsequently, Angie regularly probes each peer of the upstream group according to the parameters configured here.

A peer's probe is passed if the request to the peer succeeds, considering all parameter settings of the `upstream_probe` directive and the settings that control how upstreams are used by the directive's `location` context. This includes the *proxy_next_upstream* and *uwsgi_next_upstream* directives, etc.; also, *proxy_set_header* and so on.

To make use of the probes, the upstream must have a shared memory zone (*zone*). One upstream may be configured with several probes.

The following parameters are accepted:

`name`	Mandatory name of the probe.
`uri`	Request URI to be added to the argument for *proxy_pass*, *uwsgi_pass*, etc. By default — `/`.
`port`	Alternative port number for the probe request.
`interval`	Interval between probes. By default — `5s`.
`method`	HTTP method of the probe. By default — `GET`.
`test`	The condition for the probe, defined as a string with variables. If variable substitution yields `""` or `"0"`, the probe is not passed.
`essential`	If set, the initial state of the peer is being checked, so the peer doesn't receive client requests until the probe is passed.
`persistent`	Setting this parameter requires enabling `essential` first; `persistent` peers that were deemed healthy prior to a *configuration reload* start receiving requests without being required to pass this probe first.
`fails`	Number of subsequent failed probes that renders the peer unhealthy. By default — 1.
`passes`	Number of subsequent passed probes that renders the peer healthy. By default — 1.
`max_body`	Maximum amount of memory for the response body. By default — `256k`.
`mode`	Probe mode, depending on the peers' health: • `always` — peers are probed regardless of their state; • `idle` — probes affect unhealthy peers and peers where `interval` has elapsed since the last client request. • `onfail` — only unhealthy peers are probed. By default — `always`.

Example:

```
upstream backend {
    zone backend 1m;

    server backend1.example.com;
    server backend2.example.com;
}

map $upstream_status $good {
    200     "1";
}

server {
    listen ...;

    location /backend {
        ...
        proxy_pass http://backend;

        upstream_probe backend_probe
            uri=/probe
            port=10004
            interval=5s
            test=$good
            essential
            persistent
            fails=3
            passes=3
            max_body=10m
            mode=idle;
    }
```

```
}
```

Details of probe operation:

- Initially, the peer won't receive client requests until it passes *all* `essential` probes configured for it, skipping `persistent` ones if the configuration was reloaded and the peer was deemed healthy prior to that. If there are no such probes, the peer is considered healthy.
- The peer is considered unhealthy and won't receive client requests, if *any* of the probes configured for it hits `fails` or the peer reaches *max_fails*.
- For an unhealthy peer to be considered healthy again, *all* probes configured for it must reach their respective `passes`; after that, *max_fails* is also considered.

Built-in Variables

The `http_upstream` module supports the following built-in variables:

`$upstream_probe` (PRO)

Name of the currently active *upstream_probe*.

`$upstream_probe_body` (PRO)

Peer response body, received during an *upstream_probe*; its size is limited by `max_body`.

UserID

The module sets cookies suitable for client identification. Received and set cookies can be logged using the built-in variables *$uid_got* and *$uid_set*. This module is compatible with the mod_uid module for Apache.

Configuration Example

```
userid          on;
userid_name     uid;
userid_domain   example.com;
userid_path     /;
userid_expires  365d;
userid_p3p      'policyref="/w3c/p3p.xml", CP="CUR ADM OUR NOR STA NID"';
```

Directives

userid

Syntax	`userid on | v1 | log | off;`
Default	`userid off;`
Context	http, server, location

Enables or disables setting cookies and logging the received cookies:

`on`	enables the setting of version 2 cookies and logging of the received cookies;
`v1`	enables the setting of version 1 cookies and logging of the received cookies;
`log`	disables the setting of cookies, but enables logging of the received cookies;
`off`	disables the setting of cookies and logging of the received cookies.

userid_domain

Syntax	`userid_domain` *name* | `none`;
Default	`userid_domain none;`
Context	http, server, location

Defines a domain for which the cookie is set. The `none` parameter disables setting of a domain for the cookie.

userid_expires

Syntax	`userid_expires` *time* | `max` | `off`;
Default	`userid_expires off;`
Context	http, server, location

Sets a time during which a browser should keep the cookie. The parameter max will cause the cookie to expire on "31 Dec 2037 23:55:55 GMT". The parameter `off` will cause the cookie to expire at the end of a browser session.

userid_flags

Syntax	`userid_flags off` | *flag* ...;
Default	`userid_flags off;`
Context	http, server, location

If the parameter is not `off`, defines one or more additional flags for the cookie: `secure`, `httponly`, `samesite=strict`, `samesite=lax`, `samesite=none`.

userid_mark

Syntax	`userid_mark` *letter* | *digit* | = | `off`;
Default	`userid_mark off;`
Context	http, server, location

If the parameter is not `off`, enables the cookie marking mechanism and sets the character used as a mark. This mechanism is used to add or change *userid_p3p* and/or a cookie expiration time while preserving the client identifier. A mark can be any letter of the English alphabet (case-sensitive), digit, or the "=" character.

If the mark is set, it is compared with the first padding symbol in the base64 representation of the client identifier passed in a cookie. If they do not match, the cookie is resent with the specified mark, expiration time, and "P3P" header.

userid_name

Syntax	`userid_name` *name*;
Default	`userid_name uid;`
Context	http, server, location

Sets the cookie name.

userid_p3p

Syntax	`userid_p3p` *string* | `none`;
Default	`userid_p3p none;`
Context	http, server, location

Sets a value for the "P3P" header field that will be sent along with the cookie. If the directive is set to the special value `none`, the "P3P" header will not be sent in a response.

userid_path

Syntax	`userid_path` *path*;
Default	`userid_path /;`
Context	http, server, location

Defines a path for which the cookie is set.

userid_service

Syntax	`userid_service` *number*;
Default	`userid_service IP address of the server;`
Context	http, server, location

If identifiers are issued by multiple servers (services), each service should be assigned its own `number` to ensure that client identifiers are unique. For version 1 cookies, the default value is zero. For version 2 cookies, the default value is the number composed from the last four octets of the server's IP address.

Built-in Variables

`$uid_got`

The cookie name and received client identifier.

`$uid_reset`

If the variable is set to a non-empty string that is not `0`, the client identifiers are reset. The special value `log` additionally leads to the output of messages about the reset identifiers to the *error_log*.

`$uid_set`

The cookie name and sent client identifier.

uWSGI

Allows passing requests to a uwsgi server.

Configuration Example

```
location / {
    include    uwsgi_params;
    uwsgi_pass localhost:9000;
}
```

Directives

uwsgi_bind

Syntax	`uwsgi_bind` *address* [`transparent`] | `off`;
Default	—
Context	http, server, location

Makes outgoing connections to a uwsgi server originate from the specified local IP address with an optional port. Parameter value can contain variables. The special value `off` cancels the effect of the *uwsgi_bind* directive inherited from the previous configuration level, which allows the system to auto-assign the local IP address and port.

The `transparent` parameter allows outgoing connections to a uwsgi server originate from a non-local IP address, for example, from a real IP address of a client:

```
uwsgi_bind $remote_addr transparent;
```

In order for this parameter to work, it is usually necessary to run Angie worker processes with the *superuser* privileges. On Linux it is not required as if the `transparent` parameter is specified, worker processes inherit the *CAP_NET_RAW* capability from the master process.

Important

It is necessary to configure kernel routing table to intercept network traffic from the uwsgi server.

uwsgi_buffer_size

Syntax	`uwsgi_buffer_size` *size*;
Default	`uwsgi_buffer_size 4k|8k;`
Context	http, server, location

Sets the size of the buffer used for reading the first part of the response received from the uwsgi server. This part usually contains a small response header. By default, the buffer size is equal to one memory page. This is either 4K or 8K, depending on a platform. It can be made smaller, however.

uwsgi_buffering

Syntax	`uwsgi_buffering` *size*;
Default	`uwsgi_buffering on;`
Context	http, server, location

Enables or disables buffering of responses from the uwsgi server.

`on`	Angie receives a response from the uwsgi server as soon as possible, saving it into the buffers set by the *uwsgi_buffer_size* and *uwsgi_buffers* directives. If the whole response does not fit into memory, a part of it can be saved to a *temporary file* on the disk. Writing to temporary files is controlled by the *uwsgi_max_temp_file_size* and *uwsgi_temp_file_write_size* directives.
`off`	The response is passed to a client synchronously, immediately as it is received. Angie will not try to read the whole response from the uwsgi server. The maximum size of the data that Angie can receive from the server at a time is set by the *uwsgi_buffer_size* directive.

Buffering can also be enabled or disabled by passing "yes" or "no" in the "X-Accel-Buffering" response header field. This capability can be disabled using the *uwsgi_ignore_headers* directive.

uwsgi_buffers

Syntax	`uwsgi_buffers` *number size*;
Default	`uwsgi_buffers 8 4k	8k;`
Context	http, server, location

Sets the number and size of the buffers used for reading a response from the uwsgi server, for a single connection.

By default, the buffer size is equal to one memory page. This is either 4K or 8K, depending on a platform.

uwsgi_busy_buffers_size

Syntax	`uwsgi_busy_buffers_size` *size*;
Default	`uwsgi_busy_buffers_size 8k	16k;`
Context	http, server, location

When *buffering* of responses from the uwsgi server is enabled, limits the total size of buffers that can be busy sending a response to the client while the response is not yet fully read. In the meantime, the rest of the buffers can be used for reading the response and, if needed, buffering part of the response to a temporary file.

By default, size is limited by the size of two buffers set by the *uwsgi_buffer_size* and *uwsgi_buffers* directives.

uwsgi_cache

Syntax	`uwsgi_cache` *zone*	`off`;
Default	`uwsgi_cache off;`
Context	http, server, location

Defines a shared memory zone used for caching. The same zone can be used in several places. Parameter value can contain variables.

`off`	disables caching inherited from the previous configuration level.

uwsgi_cache_background_update

Syntax	`uwsgi_cache_background_update on`	`off`;
Default	`uwsgi_cache_background_update off;`
Context	http, server, location

Allows starting a background subrequest to update an expired cache item, while a stale cached response is returned to the client.

Attention

Note that it is necessary to *allow* the usage of a stale cached response when it is being updated.

uwsgi_cache_bypass

Syntax	`uwsgi_cache_bypass ...;`
Default	—
Context	http, server, location

Defines conditions under which the response will not be taken from a cache. If at least one value of the string parameters is not empty and is not equal to "0" then the response will not be taken from the cache:

```
uwsgi_cache_bypass $cookie_nocache $arg_nocache$arg_comment;
uwsgi_cache_bypass $http_pragma    $http_authorization;
```

Can be used along with the *uwsgi_no_cache* directive.

uwsgi_cache_key

Syntax	`uwsgi_cache_key` *string*;
Default	—
Context	http, server, location

Defines a key for caching, for example

```
uwsgi_cache_key localhost:9000$request_uri;
```

uwsgi_cache_lock

Syntax	`uwsgi_cache_lock on | off;`
Default	`uwsgi_cache_lock off;`
Context	http, server, location

When enabled, only one request at a time will be allowed to populate a new cache element identified according to the *uwsgi_cache_key* directive by passing a request to a uwsgi server. Other requests of the same cache element will either wait for a response to appear in the cache or the cache lock for this element to be released, up to the time set by the *uwsgi_cache_lock_timeout* directive.

uwsgi_cache_lock_age

Syntax	`uwsgi_cache_lock_age` *time*;
Default	`uwsgi_cache_lock_age 5s;`
Context	http, server, location

If the last request passed to the uwsgi server for populating a new cache element has not completed for the specified time, one more request may be passed to the uwsgi server.

uwsgi_cache_lock_timeout

Syntax	`uwsgi_cache_lock_timeout` *time*;
Default	`uwsgi_cache_lock_timeout 5s;`
Context	http, server, location

Sets a timeout for *uwsgi_cache_lock*. When the time expires, the request will be passed to the uwsgi server, however, the response will not be cached.

uwsgi_cache_max_range_offset

Syntax	`uwsgi_cache_max_range_offset` *number*;
Default	—
Context	http, server, location

Sets an offset in bytes for byte-range requests. If the range is beyond the offset, the range request will be passed to the uwsgi server and the response will not be cached.

uwsgi_cache_methods

Syntax	`uwsgi_cache_methods GET | HEAD | POST ...;`
Default	`uwsgi_cache_methods GET HEAD;`
Context	http, server, location

If the client request method is listed in this directive then the response will be cached. "GET" and "HEAD" methods are always added to the list, though it is recommended to specify them explicitly. See also the *uwsgi_no_cache* directive.

uwsgi_cache_min_uses

Syntax	`uwsgi_cache_min_uses` *number*;
Default	`uwsgi_cache_min_uses 1;`
Context	http, server, location

Sets the number of requests after which the response will be cached.

uwsgi_cache_path

Syntax	`uwsgi_cache_path` *path* [`levels=`*levels*] [`use_temp_path=on` | `off`] `keys_zone=`*name:size* [`inactive=`*time*] [`max_size=`*size*] [`min_free=`*size*] [`manager_files=`*number*] [`manager_sleep=`*time*] [`manager_threshold=`*time*] [`loader_files=`*number*] [`loader_sleep=`*time*] [`loader_threshold=`*time*];
Default	—
Context	http

Sets the path and other parameters of a cache. Cache data are stored in files. The file name in a cache is a result of applying the MD5 function to the *cache key*.

The `levels` parameter defines hierarchy levels of a cache: from 1 to 3, each level accepts values 1 or 2. For example, in the following configuration:

```
uwsgi_cache_path /data/angie/cache levels=1:2 keys_zone=one:10m;
```

file names in a cache will look like this:

```
/data/angie/cache/c/29/b7f54b2df7773722d382f4809d65029c
```

A cached response is first written to a temporary file, and then the file is renamed. Temporary files and the cache can be put on different file systems. However, be aware that in this case a file is copied across

two file systems instead of the cheap renaming operation. It is thus recommended that for any given `location` both saved files and a directory holding temporary files are put on the same file system.

The directory for temporary files is set based on the `use_temp_path` parameter.

`on`	If this parameter is omitted or set to the value on, the directory set by the *uwsgi_temp_path* directive for the given `location` will be used.
`off`	Temporary files will be put directly in the cache directory.

In addition, all active keys and information about data are stored in a shared memory zone, whose name and size are configured by the `keys_zone` parameter. One megabyte zone can store about 8 thousand keys.

Cached data that are not accessed during the time specified by the `inactive` parameter get removed from the cache regardless of their freshness.

By default, `inactive` is set to 10 minutes.

A special **cache manager** process monitors the maximum cache size and the minimum amount of free space on the file system with cache and when the size is exceeded or there is not enough free space, it removes the least recently used data. The data is removed in iterations.

`max_size`	maximum uwsgi_cache size
`min_free`	minimum amount of free space on the file system with cache
`manager_files`	limits the number of items to be deleted during one iteration By default, `100`
`manager_threshol`	limits the duration of one iteration By default, `200` milliseconds
`manager_sleep`	configures a pause between interactions By default, `50` milliseconds

A minute after Angie starts, the special **cache loader** process is activated. It loads information about previously cached data stored on file system into a cache zone. The loading is also done in iterations.

`loader_files`	limits the number of items to load during one iteration By default, `100`
`loader_threshold`	limits the duration of one iteration By default, `200` milliseconds
`loader_sleep`	configures a pause between interactions By default, `50` milliseconds

uwsgi_cache_revalidate

Syntax	`uwsgi_cache_revalidate on | off;`
Default	`uwsgi_cache_revalidate off;`
Context	http, server, location

Enables revalidation of expired cache items using conditional requests with the "If-Modified-Since" and "If-None-Match" header fields.

uwsgi_cache_use_stale

Syntax	`uwsgi_cache_use_stale error | timeout | invalid_header | updating | http_500 | http_503 | http_403 | http_404 | http_429 | off ...;`
Default	`uwsgi_cache_use_stale off;`
Context	http, server, location

Determines in which cases a stale cached response can be used during communication with the uwsgi server. The directive's parameters match the parameters of the *uwsgi_next_upstream* directive.

`error`	permits using a stale cached response if a uwsgi server to process a request cannot be selected.
`updating`	additional parameter, permits using a stale cached response if it is currently being updated. This allows minimizing the number of accesses to uwsgi servers when updating cached data.

Using a stale cached response can also be enabled directly in the response header for a specified number of seconds after the response became stale:

- The stale-while-revalidate extension of the "Cache-Control" header field permits using a stale cached response if it is currently being updated.
- The stale-if-error extension of the "Cache-Control" header field permits using a stale cached response in case of an error.

Note

This has lower priority than using the directive parameters.

To minimize the number of accesses to uwsgi servers when populating a new cache element, the *uwsgi_cache_lock* directive can be used.

uwsgi_cache_valid

Syntax	`uwsgi_cache_valid` [*code* ...] *time*;
Default	—
Context	http, server, location

Sets caching time for different response codes. For example, the following directives

```
uwsgi_cache_valid 200 302 10m;
uwsgi_cache_valid 404      1m;
```

set 10 minutes of caching for responses with codes 200 and 302 and 1 minute for responses with code 404.

If only caching time is specified

```
uwsgi_cache_valid 5m;
```

then only 200, 301, and 302 responses are cached.

In addition, the `any` parameter can be specified to cache any responses:

```
uwsgi_cache_valid 200 302 10m;
uwsgi_cache_valid 301      1h;
uwsgi_cache_valid any      1m;
```

Note

Parameters of caching can also be set directly in the response header. This has higher priority than setting of caching time using the directive

- The "X-Accel-Expires" header field sets caching time of a response in seconds. The zero value disables caching for a response. If the value starts with the @ prefix, it sets an absolute time in seconds since Epoch, up to which the response may be cached.
- If the header does not include the "X-Accel-Expires" field, parameters of caching may be set in the header fields "Expires" or "Cache-Control".
- If the header includes the "Set-Cookie" field, such a response will not be cached.
- If the header includes the "Vary" field with the special value "*", such a response will not be cached. If the header includes the "Vary" field with another value, such a response will be cached taking into account the corresponding request header fields.

Processing of one or more of these response header fields can be disabled using the *uwsgi_ignore_headers* directive.

uwsgi_connect_timeout

Syntax	`uwsgi_connect_timeout` *time*;
Default	`uwsgi_connect_timeout 60s;`
Context	http, server, location

Defines a timeout for establishing a connection with a uwsgi server. It should be noted that this timeout cannot usually exceed 75 seconds.

uwsgi_connection_drop

Syntax	`uwsgi_connection_drop` *time* | `on` | `off`;
Default	`uwsgi_connection_drop off;`
Context	http, server, location

Enables termination of all connections to the proxied server after it has been removed from the group or marked as permanently unavailable by a *reresolve* process or the *API command* `DELETE`.

A connection is terminated when the next read or write event is processed for either the client or the proxied server.

Setting *time* enables a connection termination *timeout*; with `on` set, connections are dropped immediately.

uwsgi_force_ranges

Syntax	`uwsgi_force_ranges off;`
Default	`uwsgi_force_ranges off;`
Context	http, server, location

Enables byte-range support for both cached and uncached responses from the uwsgi server regardless of the "Accept-Ranges" field in these responses.

uwsgi_hide_header

Syntax	`uwsgi_hide_header` *field*;
Default	—
Context	http, server, location

By default, Angie does not pass the header fields "Status" and "X-Accel-..." from the response of a uwsgi server to a client. The *uwsgi_hide_header* directive sets additional fields that will not be passed. If, on the contrary, the passing of fields needs to be permitted, the *uwsgi_pass_header* directive can be used.

uwsgi_ignore_client_abort

Syntax	`uwsgi_ignore_client_abort on | off;`
Default	`uwsgi_ignore_client_abort off;`
Context	http, server, location

Determines whether the connection with a uwsgi server should be closed when a client closes the connection without waiting for a response.

uwsgi_ignore_headers

Syntax	`uwsgi_ignore_headers` *field* ...;
Default	—
Context	http, server, location

Disables processing of certain response header fields from the uwsgi server. The following fields can be ignored: "X-Accel-Redirect", "X-Accel-Expires", "X-Accel-Limit-Rate", "X-Accel-Buffering", "X-Accel-Charset", "Expires", "Cache-Control", "Set-Cookie", and "Vary".

If not disabled, processing of these header fields has the following effect:

- "X-Accel-Expires", "Expires", "Cache-Control", "Set-Cookie" and "Vary" set the parameters of response *caching*;
- "X-Accel-Redirect" performs an *internal* redirect to the specified URI;
- "X-Accel-Limit-Rate" sets the *rate limit* for transmission of a response to a client;
- "X-Accel-Buffering" enables or disables *buffering* of a response;
- "X-Accel-Charset" sets the desired *charset* of a response.

uwsgi_intercept_errors

Syntax	`uwsgi_intercept_errors on | off;`
Default	`uwsgi_intercept_errors off;`
Context	http, server, location

Determines whether uwsgi server responses with codes greater than or equal to 300 should be passed to a client or be intercepted and redirected to Angie for processing with the *error_page* directive.

uwsgi_limit_rate

Syntax	`uwsgi_limit_rate` *rate*;
Default	`uwsgi_limit_rate 0;`
Context	http, server, location

Limits the speed of reading the response from the uwsgi server. The *rate* is specified in bytes per second and can contain variables.

0	disables rate limiting

Note

The limit is set per a request, and so if Angie simultaneously opens two connections to the uwsgi server, the overall rate will be twice as much as the specified limit. The limitation works only if *buffering* of responses from the uwsgi server is enabled.

uwsgi_max_temp_file_size

Syntax	`uwsgi_max_temp_file_size` *size*;
Default	`uwsgi_max_temp_file_size 1024m;`
Context	http, server, location

When *buffering* of responses from the uwsgi server is enabled, and the whole response does not fit into the buffers set by the *uwsgi_buffer_size* and *uwsgi_buffers* directives, a part of the response can be saved to a temporary file. This directive sets the maximum size of the temporary file. The size of data written to the temporary file at a time is set by the *uwsgi_temp_file_write_size* directive.

0	disables buffering of responses to temporary files

Note

This restriction does not apply to responses that will be cached or *stored on disk*.

uwsgi_modifier1

Syntax	`uwsgi_modifier1` *size*;
Default	`uwsgi_modifier1 0;`
Context	http, server, location

Sets the value of the modifier1 field in the uwsgi packet header.

uwsgi_modifier2

Syntax	`uwsgi_modifier2` *size*;
Default	`uwsgi_modifier2 0;`
Context	http, server, location

Sets the value of the modifier2 field in the uwsgi packet header.

uwsgi_next_upstream

Syntax	`uwsgi_next_upstream error | timeout | invalid_header | http_500 | http_503 | http_403 | http_404 | http_429 | non_idempotent | off ...;`
Default	`uwsgi_next_upstream error timeout;`
Context	http, server, location

Specifies in which cases a request should be passed to the next server in the *upstream pool*:

`error`	an error occurred while establishing a connection with the server, passing a request to it, or reading the response header;
`timeout`	a timeout has occurred while establishing a connection with the server, passing a request to it, or reading the response header;
`invalid_header`	a server returned an empty or invalid response;
`http_500`	a server returned a response with the code 500;
`http_503`	a server returned a response with the code 503;
`http_403`	a server returned a response with the code 403;
`http_404`	a server returned a response with the code 404;
`http_429`	a server returned a response with the code 429;
`non_idempotent`	normally, requests with a non-idempotent method (`POST`, `LOCK`, `PATCH`) are not passed to the next server if a request has been sent to an upstream server; enabling this option explicitly allows retrying such requests;
`off`	disables passing a request to the next server.

Note

One should bear in mind that passing a request to the next server is only possible if nothing has been sent to a client yet. That is, if an error or timeout occurs in the middle of the transferring of a response, fixing this is impossible.

The directive also defines what is considered an *unsuccessful attempt* of communication with a server.

`error`
`timeout`
`invalid_header`
	always considered unsuccessful attempts, even if they are not specified in the directive

`http_500`
`http_503`
`http_429`
	considered unsuccessful attempts only if they are specified in the directive

`http_403` `http_404`	never considered unsuccessful attempts

Passing a request to the next server can be limited by the *number of tries* and by *time*.

uwsgi_next_upstream_timeout

Syntax	`uwsgi_next_upstream_timeout` *time*;
Default	`uwsgi_next_upstream_timeout 0;`
Context	http, server, location

Limits the time during which a request can be passed to the *next* server.

0	turns off this limitation

uwsgi_next_upstream_tries

Syntax	`uwsgi_next_upstream_tries` *number*;
Default	`uwsgi_next_upstream_tries 0;`
Context	http, server, location

Limits the number of possible tries for passing a request to the *next* server.

0	turns off this limitation

uwsgi_no_cache

Syntax	`uwsgi_no_cache` *string* ...;
Default	—
Context	http, server, location

Defines conditions under which the response will not be saved to a cache. If at least one value of the string parameters is not empty and is not equal to "0" then the response will not be saved:

```
uwsgi_no_cache $cookie_nocache $arg_nocache$arg_comment;
uwsgi_no_cache $http_pragma    $http_authorization;
```

Can be used along with the *uwsgi_cache_bypass* directive.

uwsgi_param

Syntax	`uwsgi_param` *parameter value* [`if_not_empty`];
Default	—
Context	http, server, location

Sets a parameter that should be passed to the uwsgi server. The value can contain text, variables, and their combination. These directives are inherited from the previous configuration level if and only if there are no uwsgi_param directives defined on the current level.

Standard CGI environment variables should be provided as uwsgi headers, see the *uwsgi_params* file provided in the distribution:

```
location / {
    include uwsgi_params;
#    ...
}
```

If the directive is specified with `if_not_empty` then such a parameter will be passed to the server only if its value is not empty:

```
uwsgi_param HTTPS $https if_not_empty;
```

uwsgi_pass

Syntax	`uwsgi_pass` [*protocol*://] *address*;
Default	—
Context	location, if in location

Sets the protocol and address of a uwsgi server and an optional URI to which a location should be mapped. As a protocol, `uwsgi` or `suwsgi` (secured `uwsgi`, `uwsgi` over SSL) can be specified. The address can be specified as a domain name or IP address, and a port:

```
uwsgi_pass localhost:9000;
uwsgi_pass uwsgi://localhost:9000;
uwsgi_pass suwsgi://[2001:db8::1]:9090;
```

or as a UNIX domain socket path:

```
uwsgi_pass unix:/tmp/uwsgi.socket;
```

If a domain name resolves to several addresses, all of them will be used in a round-robin fashion. In addition, an address can be specified as a *server group*. If a group is used, you cannot specify the port with it; instead, specify the port for each server within the group individually.

Parameter value can contain variables. In this case, if an address is specified as a domain name, the name is searched among the described server groups, and, if not found, is determined using a *resolver*.

uwsgi_pass_header

Syntax	`uwsgi_pass_header` *field* ...;
Default	—
Context	http, server, location

Permits passing *otherwise disabled* header fields from a uwsgi server to a client.

uwsgi_pass_request_body

Syntax	`uwsgi_pass_request_body on` | `off`;
Default	`uwsgi_pass_request_body on;`
Context	http, server, location

Indicates whether the original request body is passed to the uwsgi server. See also the *uwsgi_pass_request_headers* directive.

uwsgi_pass_request_headers

Syntax	`uwsgi_pass_request_headers on` | `off`;
Default	`uwsgi_pass_request_headers on;`
Context	http, server, location

Indicates whether the header fields of the original request are passed to the uwsgi server. See also the *uwsgi_pass_request_body* directives.

uwsgi_read_timeout

Syntax	`uwsgi_read_timeout` *time*;
Default	`uwsgi_read_timeout 60s;`
Context	http, server, location

Defines a timeout for reading a response from the uwsgi server. The timeout is set only between two successive read operations, not for the transmission of the whole response. If the uwsgi server does not transmit anything within this time, the connection is closed.

uwsgi_request_buffering

Syntax	`uwsgi_request_buffering on` | `off`;
Default	`uwsgi_request_buffering on;`
Context	http, server, location

Enables or disables buffering of a client request body.

`on`	the entire request body is *read* from the client before sending the request to a uwsgi server.
`off`	the request body is sent to the uwsgi server immediately as it is received. In this case, the request cannot be passed to the *next server* if Angie already started sending the request body.

When HTTP/1.1 chunked transfer encoding is used to send the original request body, the request body will be buffered regardless of the directive value.

uwsgi_send_timeout

Syntax	`uwsgi_send_timeout` *time*;
Default	`uwsgi_send_timeout 60s;`
Context	http, server, location

Sets a timeout for transmitting a request to the uwsgi server. The timeout is set only between two successive write operations, not for the transmission of the whole request. If the uwsgi server does not receive anything within this time, the connection is closed.

uwsgi_socket_keepalive

Syntax	`uwsgi_socket_keepalive on` | `off`;
Default	`uwsgi_socket_keepalive off;`
Context	http, server, location

Configures the "TCP keepalive" behavior for outgoing connections to a uwsgi server.

`""`	By default, the operating system's settings are in effect for the socket.
`on`	The *SO_KEEPALIVE* socket option is turned on for the socket.

uwsgi_ssl_certificate

Syntax	`uwsgi_ssl_certificate` *file*;
Default	—
Context	http, server, location

Specifies a file with the certificate in the PEM format used for authentication to a secured uwsgi server. Variables can be used in the file name.

uwsgi_ssl_certificate_cache

Syntax	`uwsgi_ssl_certificate_cache off`; `uwsgi_ssl_certificate_cache max=`*N* [`inactive=`*time*] [`valid=`*time*];
Default	`uwsgi_ssl_certificate_cache off;`
Context	http, server, location

Defines a cache that stores *SSL certificates* and *secret keys* specified using variables.

The directive supports the following parameters:

- `max` — sets the maximum number of elements in the cache. When the cache overflows, the least recently used (LRU) elements are removed.
- `inactive` — defines the time after which an element is removed if it has not been accessed. The default is 10 seconds.
- `valid` — defines the time during which a cached element is considered valid and can be reused. The default is 60 seconds. After this period, certificates are reloaded or revalidated.
- `off` — disables the cache.

Example:

```
uwsgi_ssl_certificate       $uwsgi_ssl_server_name.crt;
uwsgi_ssl_certificate_key   $uwsgi_ssl_server_name.key;
uwsgi_ssl_certificate_cache max=1000 inactive=20s valid=1m;
```

uwsgi_ssl_certificate_key

Syntax	`uwsgi_ssl_certificate_key` *file*;
Default	—
Context	http, server, location

Specifies a file with the secret key in the PEM format used for authentication to a secured uwsgi server.

The value "engine:*name*:id" can be specified instead of the file, which loads a secret key with a specified id from the OpenSSL engine name. Variables can be used in the file name.

uwsgi_ssl_ciphers

Syntax	`uwsgi_ssl_ciphers` *ciphers*;
Default	`uwsgi_ssl_ciphers DEFAULT;`
Context	http, server, location

Specifies the enabled ciphers for requests to a secured uwsgi server. The ciphers are specified in the format understood by the OpenSSL library.

The list of ciphers depends on the version of OpenSSL installed. The full list can be viewed using the `openssl ciphers` command.

Attention

The `uwsgi_ssl_ciphers` directive does *not* configure ciphers for TLS 1.3 when using OpenSSL. To tune TLS 1.3 ciphers with OpenSSL, use the *uwsgi_ssl_conf_command* directive, which was added to support advanced SSL configuration.

- In LibreSSL, TLS 1.3 ciphers *can* be configured using `uwsgi_ssl_ciphers`.
- In BoringSSL, TLS 1.3 ciphers cannot be configured at all.

uwsgi_ssl_conf_command

Syntax	`uwsgi_ssl_conf_command` *name value*;
Default	—
Context	http, server, location

Sets arbitrary OpenSSL configuration commands when establishing a connection with the secured uwsgi server.

Important

The directive is supported when using OpenSSL 1.0.2 or higher. To configure TLS 1.3 ciphers with OpenSSL, use the `ciphersuites` command.

Several *uwsgi_ssl_conf_command* directives can be specified on the same level. These directives are inherited from the previous configuration level if and only if there are no *uwsgi_ssl_conf_command* directives defined on the current level.

Caution

Note that configuring OpenSSL directly might result in unexpected behavior.

uwsgi_ssl_crl

Syntax	`uwsgi_ssl_crl` *file*;
Default	—
Context	http, server, location

Specifies a file with revoked certificates (CRL) in the PEM format used to *verify* the certificate of the secured uwsgi server.

uwsgi_ssl_name

Syntax	`uwsgi_ssl_name` *name*;
Default	`uwsgi_ssl_name` `` `host from uwsgi_pass;` ``
Context	http, server, location

Allows overriding the server name used to *verify* the certificate of the secured uwsgi server and to be *passed through SNI* when establishing a connection with the secured uwsgi server.

By default, the host part of the *uwsgi_pass* URL is used.

uwsgi_ssl_password_file

Syntax	`uwsgi_ssl_password_file` *file*;
Default	—
Context	http, server, location

Specifies a file with passphrases for *secret keys* where each passphrase is specified on a separate line. Passphrases are tried in turn when loading the key.

uwsgi_ssl_protocols

Syntax	`uwsgi_ssl_protocols` [SSLv2] [SSLv3] [TLSv1] [TLSv1.1] [TLSv1.2] [TLSv1.3];
Default	`uwsgi_ssl_protocols TLSv1.2 TLSv1.3;`
Context	http, server, location

Changed in version 1.2.0: `TLSv1.3` parameter added to default set.

Enables the specified protocols for requests to a secured uwsgi server.

uwsgi_ssl_server_name

Syntax	`uwsgi_ssl_server_name on` | `off`;
Default	`uwsgi_ssl_server_name off;`
Context	http, server, location

Enables or disables passing the server name set by the *uwsgi_ssl_name* directive via the Server Name Indication TLS extension (SNI, RFC 6066) while establishing a connection with the secured uwsgi server.

uwsgi_ssl_session_reuse

Syntax	`uwsgi_ssl_session_reuse on` | `off`;
Default	`uwsgi_ssl_session_reuse on;`
Context	http, server, location

Determines whether SSL sessions can be reused when working with the uwsgi server. If the errors "*SSL3_GET_FINISHED:digest check failed*" appear in the logs, try disabling *session reuse*.

uwsgi_ssl_trusted_certificate

Syntax	`uwsgi_ssl_trusted_certificate` *file*;
Default	—
Context	http, server, location

Specifies a file with trusted CA certificates in the PEM format used to *verify* the certificate of the secured uwsgi server.

uwsgi_ssl_verify

Syntax	`uwsgi_ssl_verify on	off;`
Default	`uwsgi_ssl_verify off;`
Context	http, server, location

Enables or disables verification of the secured uwsgi server certificate.

uwsgi_ssl_verify_depth

Syntax	`uwsgi_ssl_verify_depth` *number*;
Default	`uwsgi_ssl_verify_depth 1;`
Context	http, server, location

Sets the verification depth in the secured uwsgi server certificates chain.

uwsgi_store

Syntax	`uwsgi_store on	off |` *string*;
Default	`uwsgi_store off;`
Context	http, server, location

Enables saving of files to a disk.

`on`	saves files with paths corresponding to the directives *alias* or *root*
`off`	disables saving of files

The file name can be set explicitly using the `string` with variables:

```
uwsgi_store /data/www$original_uri;
```

The modification time of files is set according to the received "Last-Modified" response header field. The response is first written to a temporary file, and then the file is renamed. Temporary files and the persistent store can be put on different file systems. However, be aware that in this case a file is copied across two file systems instead of the cheap renaming operation. It is thus recommended that for any given `location` both saved files and a directory holding temporary files, set by the *uwsgi_temp_path* directive, are put on the same file system.

This directive can be used to create local copies of static unchangeable files, e.g.:

```
location /images/ {
    root               /data/www;
    error_page         404 = /fetch$uri;
}

location /fetch/ {
    internal;

    uwsgi_pass         backend:9000;
    ...

    uwsgi_store        on;
    uwsgi_store_access user:rw group:rw all:r;
    uwsgi_temp_path    /data/temp;
```

```
    alias             /data/www/;
}
```

uwsgi_store_access

Syntax	`uwsgi_store_access` *users:permissions ...*;
Default	`uwsgi_store_access user:rw;`
Context	http, server, location

Sets access permissions for newly created files and directories, e.g.:

```
uwsgi_store_access user:rw group:rw all:r;
```

If any `group` or `all` access permissions are specified then user permissions may be omitted:

```
uwsgi_store_access group:rw all:r;
```

uwsgi_temp_file_write_size

Syntax	`uwsgi_temp_file_write_size` *size*;
Default	`uwsgi_temp_file_write_size 8k|16k;`
Context	http, server, location

Limits the size of data written to a temporary file at a time, when buffering of responses from the uwsgi server to temporary files is enabled. By default, size is limited by two buffers set by the *uwsgi_buffer_size* and *uwsgi_buffers* directives. The maximum size of a temporary file is set by the *uwsgi_max_temp_file_size* directive.

uwsgi_temp_path

Syntax	`uwsgi_temp_path` *path* [*level1* [*level2* [*level3*]]]`;`
Default	`uwsgi_temp_path uwsgi_temp;` (the path depends on the `--http-uwsgi-temp-path` build option)
Context	http, server, location

Defines a directory for storing temporary files with data received from uwsgi servers. Up to three-level subdirectory hierarchy can be used underneath the specified directory. For example, in the following configuration

```
uwsgi_temp_path /spool/angie/uwsgi_temp 1 2;
```

a temporary file might look like this:

```
/spool/angie/uwsgi_temp/7/45/00000123457
```

See also the *use_temp_path* parameter of the *uwsgi_cache_path* directive.

HTTP/2

Provides support for HTTP/2.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-http_v2_module` build option.

In packages and images from our repos, the module is included in the build.

Configuration Example

```
server {
    listen 443 ssl;

    http2 on;

    ssl_certificate server.crt;
    ssl_certificate_key server.key;
}
```

Important

Note that accepting HTTP/2 connections over TLS requires the "Application-Layer Protocol Negotiation" (ALPN) TLS extension support, which is available since OpenSSL version 1.0.2.

If the *ssl_prefer_server_ciphers* directive is set to the value "on", the *ciphers* should be configured to comply with RFC 9113, Appendix A black list and supported by clients.

Directives

http2

Added in version 1.2.0.

Syntax	`http2 on	off;`
Default	`http2 off;`
Context	http, server

Enables the HTTP/2 protocol.

http2_body_preread_size

Syntax	`http2_body_preread_size` *size*;
Default	—
Context	http, server

Sets the size of the buffer per each request in which the request body may be saved before it is started to be processed.

http2_chunk_size

Syntax	`http2_chunk_size` *size*;
Default	`http2_chunk_size 8k;`
Context	http, server, location

Sets the maximum size of chunks into which the response body is sliced. A too low value results in higher overhead. A too high value impairs prioritization due to HOL blocking.

http2_max_concurrent_pushes

Deprecated since version 1.2.0.

Syntax	`http2_max_concurrent_pushes` *number*;
Default	`http2_max_concurrent_pushes 10;`
Context	http, server

Limits the maximum number of concurrent *push* requests in a connection.

http2_max_concurrent_streams

Syntax	`http2_max_concurrent_streams` *number*;
Default	`http2_max_concurrent_streams 128;`
Context	http, server

Sets the maximum number of concurrent HTTP/2 streams in a connection.

http2_push

Deprecated since version 1.2.0.

Syntax	`http2_push` *uri* | `off`;
Default	`http2_push off;`
Context	http, server, location

Preemptively sends (pushes) a request to the specified uri along with the response to the original request. Only relative URIs with absolute path will be processed, for example:

```
http2_push /static/css/main.css;
```

The `uri` value can contain variables.

Several *http2_push* directives can be specified on the same configuration level. The `off` parameter cancels the effect of the *http2_push* directives inherited from the previous configuration level.

http2_push_preload

Deprecated since version 1.2.0.

Syntax	`http2_push_preload on` | `off`;
Default	`http2_push_preload off;`
Context	http, server, location

Enables automatic conversion of preload links specified in the "Link" response header fields into push requests.

http2_recv_buffer_size

Syntax	`http2_recv_buffer_size` *size*;
Default	`http2_recv_buffer_size 256k;`
Context	http

Sets the size of the per *worker* input buffer.

Built-in Variables

The *http_v2* module supports the following built-in variables:

`$http2`

negotiated protocol identifier:

`h2`	for HTTP/2 over TLS
`h2c`	for HTTP/2 over cleartext TCP
`""`	an empty string otherwise

HTTP/3

Enables HTTP/3 support for client connections, as well as for connections with proxied servers configured using the following *http_proxy* directives:

- *proxy_http3_hq*
- *proxy_http3_max_concurrent_streams*
- *proxy_http3_max_table_capacity*
- *proxy_http3_stream_buffer_size*
- *proxy_http_version*
- *proxy_pass*
- *proxy_quic_active_connection_id_limit*
- *proxy_quic_gso*
- *proxy_quic_host_key*

When building from the source code, this module isn't built by default; it should be enabled with the `--with-http_v3_module` build option.

In packages and images from our repos, the module is included in the build.

Configuration Example

```
http {
    log_format quic '$remote_addr - $remote_user [$time_local] '
                    '"$request" $status $body_bytes_sent '
                    '"$http_referer" "$http_user_agent" "$http3"';

    access_log logs/access.log quic;

    server {
        # for better compatibility it's recommended
        # to use the same port for http/3 and https
        listen 8443 quic reuseport;
        listen 8443 ssl;

        ssl_certificate     certs/example.com.crt;
        ssl_certificate_key certs/example.com.key;

        location / {
            # used to advertise the availability of HTTP/3
            add_header Alt-Svc 'h3=":8443"; ma=86400';
        }
```

```
    }
}
```

Important

Note that accepting HTTP/3 connections over TLS requires the TLSv1.3 protocol support, which is available since OpenSSL version 1.1.1.

Directives

http3

Syntax	`http3 on	off;`
Default	`http3 on;`
Context	http, server

Enables HTTP/3 protocol negotiation.

http3_hq

Syntax	`http3_hq on	off;`
Default	`http3_hq off;`
Context	http, server

Enables HTTP/0.9 protocol negotiation used in QUIC interoperability tests.

Attention

Enable this mode only to run specialized tests that explicitly require it.

http3_max_concurrent_streams

Syntax	`http3_max_concurrent_streams` *number*;
Default	`http3_max_concurrent_streams 128;`
Context	http, server

Initializes HTTP/3 and QUIC settings and sets the maximum number of concurrent HTTP/3 request streams in a connection.

http3_max_table_capacity

Syntax	`http3_max_table_capacity` *number*;
Default	`http3_max_table_capacity 4096;`
Context	http, server

Sets the dynamic table capacity for server connections.

Note

A similar *proxy_http3_max_table_capacity* directive does this for proxy connections. To avoid errors, dynamic table usage is disabled when proxying with caching is enabled.

http3_stream_buffer_size

Syntax	`http3_stream_buffer_size` *size*;
Default	`http3_stream_buffer_size 64k;`
Context	http, server

Sets the size of the buffer used for reading and writing of the QUIC streams.

quic_active_connection_id_limit

Syntax	`quic_active_connection_id_limit` *number*;
Default	`quic_active_connection_id_limit 2;`
Context	http, server

Sets the QUIC *active_connection_id_limit* transport parameter value. This is the maximum number of client connection IDs which can be stored on the server.

quic_bpf

Syntax	`quic_bpf on | off;`
Default	`quic_bpf off;`
Context	main

Enables routing of QUIC packets using eBPF. When enabled, this allows supporting QUIC connection migration.

Important

The directive is only supported on Linux 5.7+.

quic_gso

Syntax	`quic_gso on | off;`
Default	`quic_gso off;`
Context	http, server

Enables sending in optimized batch mode using segmentation offloading.

Important

Optimized sending is supported only on Linux featuring *UDP_SEGMENT*.

quic_host_key

Syntax	`quic_host_key` *file*;
Default	—
Context	http, server

Sets a *file* with the secret key used to encrypt stateless reset and address validation tokens. By default, a random key is generated on each reload. Tokens generated with old keys are not accepted.

quic_retry

Syntax	`quic_retry on | off;`
Default	`quic_retry off;`
Context	http, server

Enables the QUIC Address Validation feature. This includes sending a new token in a *Retry* packet or a *NEW_TOKEN* frame and validating a token received in the *Initial* packet.

Built-in Variables

The *http_v3* module supports the following built-in variables:

`$http3`

negotiated protocol identifier:

`h3`	for HTTP/3 connections
`hq`	for hq connections
`""`	an empty string otherwise

`$quic_connection`

QUIC connection serial number

XSLT

The module is a filter that transforms XML responses using one or more XSLT stylesheets.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-http_xslt_module` build option.

In our repositories, the module is built dynamically and is available as a separate package named `angie-module-xslt` or `angie-pro-module-xslt`.

Important

This module requires the libxml2 and libxslt libraries.

Configuration Example

```
location / {
    xml_entities    /site/dtd/entities.dtd;
    xslt_stylesheet /site/xslt/one.xslt param=value;
```

```
    xslt_stylesheet /site/xslt/two.xslt;
}
```

Directives

xml_entities

Syntax	`xml_entities` *path*;
Default	—
Context	http, server, location

Specifies the DTD file that declares character entities. This file is compiled at the configuration stage. For technical reasons, the module is unable to use the external subset declared in the processed XML, so it is ignored and a specially defined file is used instead. This file should not describe the XML structure. It is enough to declare just the required character entities, for example:

```
<!ENTITY nbsp " ">
```

xslt_last_modified

Syntax	`xslt_last_modified on | off;`
Default	`xslt_last_modified off;`
Context	http, server, location

Allows preserving the "Last-Modified" header field from the original response during XSLT transformations to facilitate response caching.

By default, the header field is removed as contents of the response are modified during transformations and may contain dynamically generated elements or parts that are changed independently of the original response.

xslt_param

Syntax	`xslt_param` *parameter value*;
Default	—
Context	http, server, location

Defines the parameters for XSLT stylesheets. The value is treated as an XPath expression. The value can contain variables. To pass a string value to a stylesheet, the *xslt_string_param* directive can be used.

There could be several *xslt_param* directives. These directives are inherited from the previous configuration level if and only if there are no *xslt_param* and *xslt_string_param* directives defined on the current level.

xslt_string_param

Syntax	`xslt_string_param` *parameter value*;
Default	—
Context	http, server, location

Defines the string parameters for XSLT stylesheets. XPath expressions in the value are not interpreted. The value can contain variables.

There could be several *xslt_string_param* directives. These directives are inherited from the previous configuration level if and only if there are no *xslt_param* and *xslt_string_param* directives defined on the current level.

xslt_stylesheet

Syntax	`xslt_stylesheet` *stylesheet* [*parameter*=*value* ...];
Default	—
Context	location

Defines the XSLT stylesheet and its optional parameters. A stylesheet is compiled at the configuration stage.

Parameters can either be specified separately, or grouped in a single line using the ":" delimiter. If a parameter includes the ":" character, it should be escaped as "%3A". Also, libxslt requires to enclose parameters that contain non-alphanumeric characters into single or double quotes, for example:

```
param1='http%3A//www.example.com':param2=value2
```

The parameters description can contain variables, for example, the whole line of parameters can be taken from a single variable:

```
location / {
    xslt_stylesheet /site/xslt/one.xslt
                    $arg_xslt_params
                    param1='$value1':param2=value2
                    param3=value3;
}
```

It is possible to specify several stylesheets. They will be applied sequentially in the specified order.

xslt_types

Syntax	`xslt_types` mime-*type* ...;
Default	`xslt_types text/xml;`
Context	http, server, location

Enables transformations in responses with the specified MIME types in addition to `text/xml`. The special value "*" matches any MIME type. If the transformation result is an HTML response, its MIME type is changed to `text/html`.

The core HTTP module implements the basic functionality of an HTTP server: this includes defining server blocks, configuring locations for request routing, serving static files and controlling access, configuring redirects, supporting `keep-alive` connections, and managing request and response headers.

The other modules in this section extend this functionality, allowing you to flexibly configure and optimize the HTTP server for various scenarios and requirements.

Directives

absolute_redirect

Syntax	`absolute_redirect on` | `off`;
Default	`absolute_redirect on;`
Context	http, server, location

If disabled, redirects issued by Angie will be relative.

See also *server_name_in_redirect* and *port_in_redirect* directives.

aio

Syntax	`aio on` | `off` | `threads` [=*pool*];
Default	`aio off;`
Context	http, server, location

Enables or disables the use of asynchronous file I/O (AIO) on FreeBSD and Linux:

```
location /video/ {
  aio            on;
  output_buffers 1 64k;
}
```

On FreeBSD, AIO can be used starting from FreeBSD 4.3. Prior to FreeBSD 11.0, AIO can either be linked statically into a kernel:

```
options VFS_AIO
```

or loaded dynamically as a kernel loadable module:

```
kldload aio
```

On Linux, AIO can be used starting from kernel version 2.6.22. Also, it is necessary to enable *directio*, or otherwise reading will be blocking:

```
location /video/ {
  aio            on;
  directio       512;
  output_buffers 1 128k;
}
```

On Linux, *directio* can only be used for reading blocks that are aligned on 512-byte boundaries (or 4K for XFS). File's unaligned end is read in blocking mode. The same holds true for byte range requests and for FLV requests not from the beginning of a file: reading of unaligned data at the beginning and end of a file will be blocking.

When both AIO and *sendfile* are enabled on Linux, AIO is used for files that are larger than or equal to the size specified in the *directio* directive, while *sendfile* is used for files of smaller sizes or when *directio* is disabled:

```
location /video/ {
  sendfile       on;
  aio            on;
  directio       8m;
}
```

Finally, files can be read and *sent* using multi-threading, without blocking a worker process:

```
location /video/ {
  sendfile       on;
  aio            threads;
}
```

Read and send file operations are offloaded to threads of the specified *pool*. If the `pool` name is omitted, the pool with the name "default" is used. The pool name can also be set with variables:

```
aio threads=pool$disk;
```

By default, multi-threading is disabled, it should be enabled with the *--with-threads* configuration parameter. Currently, multi-threading is compatible only with the *epoll*, *kqueue* and *eventport* methods. Multi-threaded sending of files is only supported on Linux.

See also the *sendfile* directive.

aio_write

Syntax	`aio_write on	off;`
Default	`aio_write off;`
Context	http, server, location

If *aio* is enabled, specifies whether it is used for writing files. Currently, this only works when using *aio threads* and is limited to writing temporary files with data received from proxied servers.

alias

Syntax	`alias` *path*;
Default	—
Context	location

Defines a replacement for the specified location. For example, with the following configuration:

```
location /i/ {
  alias /data/w3/images/;
}
```

on request of `/i/top.gif`, the file */data/w3/images/top.gif* will be sent.

The `path` value can contain variables, except *$document_root* and *$realpath_root*.

If `alias` is used inside a `location` defined with a regular expression then such regular expression should contain captures and `alias` should refer to these captures, for example:

```
location ~ ^/users/(.+\.(?:gif|jpe?g|png))$ {
  alias /data/w3/images/$1;
}
```

When location matches the last part of the directive's value:

```
location /images/ {
  alias /data/w3/images/;
}
```

it is better to use the *root* directive instead:

```
location /images/ {
  root /data/w3;
}
```

auth_delay

Syntax	`auth_delay` *time*;
Default	`auth_delay 0s;`
Context	http, server, location

Delays processing of unauthorized requests with 401 response code to prevent timing attacks when access is limited by *password* or by the *result of subrequest*.

auto_redirect

Syntax	`auto_redirect` [`on` | `off` | `default`];
Default	`auto_redirect default;`
Context	http, server, location

The directive controls the *redirection* behavior when a prefix location ends with a slash:

```
location /prefix/ {
    auto_redirect on;
}
```

Here, a request for `/prefix` causes a redirect to `/prefix/`.

The value `on` explicitly enables redirection, while `off` disables it. When set to `default`, redirection is enabled only if the `location` processes requests with *api*, *proxy_pass*, *fastcgi_pass*, *uwsgi_pass*, *scgi_pass*, *memcached_pass*, or *grpc_pass*.

chunked_transfer_encoding

Syntax	`chunked_transfer_encoding on` | `off`;
Default	`chunked_transfer_encoding on;`
Context	http, server, location

Allows disabling chunked transfer encoding in HTTP/1.1. It may come in handy when using a software failing to support chunked encoding despite the standard's requirement.

client_body_buffer_size

Syntax	`client_body_buffer_size` *size*;
Default	`client_body_buffer_size 8k|16k;`
Context	http, server, location

Sets buffer size for reading client request body. In case the request body is larger than the buffer, the whole body or only its part is written to a *temporary file*. By default, buffer size is equal to two memory pages. This is 8K on x86, other 32-bit platforms, and x86-64. It is usually 16K on other 64-bit platforms.

client_body_in_file_only

Syntax	`client_body_in_file_only on` | `clean` | `off`;
Default	`client_body_in_file_only off;`
Context	http, server, location

Determines whether Angie should save the entire client request body into a file. This directive can be used during debugging, or when using the *$request_body_file* variable, or the *$r->request_body_file* method of the module *Perl*.

When set to the value `on`, temporary files are not removed after request processing.

`on`	temporary files are not removed after request processing
`clean`	will cause the temporary files left after request processing to be removed

client_body_in_single_buffer

Syntax	`client_body_in_single_buffer on | off;`
Default	`client_body_in_single_buffer off;`
Context	http, server, location

Determines whether Angie should save the entire client request body in a single buffer. The directive is recommended when using the *$request_body* variable, to save the number of copy operations involved.

client_body_temp_path

Syntax	`client_body_temp_path` *path* [*level1* [*level2* [*level3*]]];
Default	`client_body_temp_path client_body_temp;` (the path depends on the `--http-proxy-temp-path` build option)
Context	http, server, location

Defines a directory for storing temporary files holding client request bodies. Up to three-level subdirectory hierarchy can be used under the specified directory. For example, in the following configuration

```
client_body_temp_path /spool/angie/client_temp 1 2;
```

a path to a temporary file might look like this:

```
/spool/angie/client_temp/7/45/00000123457
```

client_body_timeout

Syntax	`client_body_timeout` *time;*
Default	`client_body_timeout 60s;`
Context	http, server, location

Defines a timeout for reading client request body. The timeout is set only for a period between two successive read operations, not for the transmission of the whole request body. If a client does not transmit anything within this time, the request is terminated with the 408 (Request Time-out) error.

client_header_buffer_size

Syntax	`client_header_buffer_size` *size*;
Default	`client_header_buffer_size 1k;`
Context	http, server

Sets buffer size for reading client request header. For most requests, a buffer of 1K bytes is enough. However, if a request includes long cookies, or comes from a WAP client, it may not fit into 1K. If a

request line or a request header field does not fit into this buffer then larger buffers, configured by the *large_client_header_buffers* directive, are allocated.

If the directive is specified on the *server* level, the value from the default server can be used. Details are provided in the *Virtual server selection* section.

client_header_timeout

Syntax	`client_header_timeout` *time*;
Default	`client_header_timeout 60s;`
Context	http, server

Defines a timeout for reading client request header. If a client does not transmit the entire header within this time, the request is terminated with the 408 (Request Time-out) error.

client_max_body_size

Syntax	`client_max_body_size` *size*;
Default	`client_max_body_size 1m;`
Context	http, server, location

Sets the maximum allowed size of the client request body. If the size in a request exceeds the configured value, the 413 (Request Entity Too Large) error is returned to the client. Please be aware that browsers cannot correctly display this error.

`0`	disables checking of client request body size

connection_pool_size

Syntax	`connection_pool_size` *size*;
Default	`connection_pool_size 256 | 512;`
Context	http, server, location

Allows accurate tuning of per-connection memory allocations. This directive has minimal impact on performance and should not generally be used.

By default:

`256` (bytes)	32-bit platforms
`512` (bytes)	64-bit platforms

default_type

Syntax	`default_type` mime-*type*;
Default	`default_type text/plain;`
Context	http, server, location

Defines the default MIME type of a response. Mapping of file name extensions to MIME types can be set with the *types* directive.

directio

Syntax	`directio` *size* | `off`;
Default	`directio off;`
Context	http, server, location

Enables the use of the *O_DIRECT* flag (FreeBSD, Linux), the *F_NOCACHE* flag (macOS), or the *directio()* function (Solaris), when reading files that are larger than or equal to the specified size. The directive automatically disables the use of *sendfile* for a given request. It can be useful for serving large files:

```
directio 4m;
```

or when using *aio* on Linux.

directio_alignment

Syntax	`directio_alignment` *size*;
Default	`directio_alignment 512;`
Context	http, server, location

Sets the alignment for *directio*. In most cases, a 512-byte alignment is enough. However, when using XFS under Linux, it needs to be increased to 4K.

disable_symlinks

Syntax	`disable_symlinks off;` `disable_symlinks on | if_not_owner` [`from=`*part*];
Default	`disable_symlinks off;`
Context	http, server, location

Determines how symbolic links should be treated when opening files:

`off`	Symbolic links in the pathname are allowed and not checked. This is the default behavior.
`on`	If any component of the pathname is a symbolic link, access to a file is denied.
`if_not_owner`	Access to a file is denied if any component of the pathname is a symbolic link, and the link and object that the link points to have different owners.
`from=`*part*	When checking symbolic links (parameters `on` and `if_not_owner`), all components of the pathname are normally checked. Checking of symbolic links in the initial part of the pathname may be avoided by specifying additionally the *from=part* parameter. In this case, symbolic links are checked only from the pathname component that follows the specified initial part. If the value is not an initial part of the pathname checked, the whole pathname is checked as if this parameter was not specified at all. If the value matches the whole file name, symbolic links are not checked. The parameter value can contain variables.

Example:

```
disable_symlinks on from=$document_root;
```

This directive is only available on systems that have the `openat()` and `fstatat()` interfaces. Such systems include modern versions of FreeBSD, Linux, and Solaris.

Warning

Parameters `on` and `if_not_owner` add a processing overhead.

On systems that do not support opening of directories only for search, to use these parameters it is required that worker processes have read permissions for all directories being checked.

Note

The *AutoIndex*, *Random Index* and *DAV* modules currently ignore this directive.

error_page

Syntax	`error_page` *code* ... [=[*response*]] uri;
Default	—
Context	http, server, location, if in location

Defines the URI that will be shown for the specified errors. A `uri` value can contain variables.

Example:

```
error_page 404             /404.html;
error_page 500 502 503 504 /50x.html;
```

This causes an internal redirect to the specified `uri` with the client request method changed to "GET" (for all methods other than "GET" and "HEAD").

Furthermore, it is possible to change the response code to another using the `=response` syntax, for example:

```
error_page 404 =200 /empty.gif;
```

If an error response is processed by a proxied server or a FastCGI/uwsgi/SCGI/gRPC server, and the server may return different response codes (e.g., 200, 302, 401 or 404), it is possible to respond with the code it returns:

```
error_page 404 = /404.php;
```

If there is no need to change URI and method during internal redirection it is possible to pass error processing into a named location:

```
location / {
  error_page 404 = @fallback;
}

location @fallback {
  proxy_pass http://backend;
}
```

Note

If uri processing leads to an error, the status code of the last occurred error is returned to the client.

It is also possible to use URL redirects for error processing:

```
error_page 403      http://example.com/forbidden.html;
error_page 404 =301 http://example.com/notfound.html;
```

In this case, by default, the response code 302 is returned to the client. It can only be changed to one of the redirect status codes (301, 302, 303, 307, and 308).

etag

Syntax	`etag on	off;`
Default	`etag on;`
Context	http, server, location

Enables or disables automatic generation of the "ETag" response header field for static resources.

http

Syntax	`http { ... }`
Default	—
Context	main

Provides the configuration file context in which the HTTP server directives are specified.

if_modified_since

Syntax	`if_modified_since off	exact	before;`
Default	`if_modified_since exact;`
Context	http, server, location

Specifies how to compare modification time of a response with the time in the *If-Modified-Since* request header field:

`off`	the response is always considered modified
`exact`	exact match
`before`	modification time of the response is less than or equal to the time in the *If-Modified-Since* request header field.

ignore_invalid_headers

Syntax	`ignore_invalid_headers on	off;`
Default	`ignore_invalid_headers on;`
Context	http, server

Controls whether header fields with invalid names should be ignored. Valid names are composed of English letters, digits, hyphens, and possibly underscores (as controlled by the ref:*underscores_in_headers* directive).

If the directive is specified on the `server` level, the value from the default server can be used.

internal

Syntax	`internal;`
Default	—
Context	location

Specifies that a given location can only be used for internal requests. For external requests, the client error 404 (Not Found) is returned. Internal requests are the following:

- requests redirected by the *error_page*, *index*, *random_index* and *try_files* directives;
- requests redirected by the *X-Accel-Redirect* response header field from an upstream server;
- subrequests formed by the *include virtual* command of the *http_ssi* module, by the *http_addition* module directives, and by *auth_request* and *mirror* directives;
- requests changed by the *rewrite* directive.

Example:

```
error_page 404 /404.html;

location = /404.html {
  internal;
}
```

Note

There is a limit of 10 internal redirects per request to prevent request processing cycles that can occur in incorrect configurations. If this limit is reached, the error 500 (Internal Server Error) is returned. In such cases, the *rewrite or internal redirection cycle* message can be seen in the error log.

keepalive_disable

Syntax	`keepalive_disable none` | *browser* ...;
Default	`keepalive_disable msie6;`
Context	http, server, location

Disables keep-alive connections with misbehaving browsers. The `browser` parameters specify which browsers will be affected.

`none`	enables keep-alive connections with all browsers
`msie6`	disables keep-alive connections with old versions of MSIE, once a POST request is received
`safari`	disables keep-alive connections with Safari and Safari-like browsers on macOS and macOS-like operating systems

keepalive_requests

Syntax	`keepalive_requests` *number;*
Default	`keepalive_requests 1000;`
Context	http, server, location

Sets the maximum number of requests that can be served through one keep-alive connection. After the maximum number of requests are made, the connection is closed.

Closing connections periodically is necessary to free per-connection memory allocations. Therefore, using too high maximum number of requests could result in excessive memory usage and not recommended.

keepalive_time

Syntax	`keepalive_time` *time*;
Default	`keepalive_time 1h;`
Context	http, server, location

Limits the maximum time during which requests can be processed through one keep-alive connection. After this time is reached, the connection is closed following the subsequent request processing.

keepalive_timeout

Syntax	`keepalive_timeout` *timeout* [*header_ timeout*];
Default	`keepalive_timeout 75s;`
Context	http, server, location

`timeout`	sets a timeout during which a keep-alive client connection will stay open on the server side
`0`	disables keep-alive client connections

The *optional* second parameter sets a value in the "Keep-Alive: timeout=time" response header field. Two parameters may differ.

The "Keep-Alive: timeout=time" header field is recognized by Mozilla and Konqueror. MSIE closes keep-alive connections by itself in about 60 seconds.

large_client_header_buffers

Syntax	`large_client_header_buffers` *number size*;
Default	`large_client_header_buffers 4 8k;`
Context	http, server

Sets the maximum number and size of buffers used for reading large client request header. A request line cannot exceed the size of one buffer, or the 414 (Request-URI Too Large) error is returned to the client. A request header field cannot exceed the size of one buffer as well, or the 400 (Bad Request) error is returned to the client. Buffers are allocated only on demand. By default, the buffer size is equal to 8K bytes. If after the end of request processing a connection is transitioned into the keep-alive state, these buffers are released.

If the directive is specified on the *server* level, the value from the default server can be used.

limit_except

Syntax	`limit_except` *method1* [*method2...*] { ... };
Default	—
Context	location

Limits allowed HTTP methods inside a location. The `method` can be one of the following: `GET`, `HEAD`, `POST`, `PUT`, `DELETE`, `MKCOL`, `COPY`, `MOVE`, `OPTIONS`, `PROPFIND`, `PROPPATCH`, `LOCK`, `UNLOCK` or `PATCH`. Allowing the `GET` method also enabled the `HEAD` method. Access to other methods can be limited using the directives from *Access* and *Auth Basic* modules:

```
limit_except GET {
  allow 192.168.1.0/32;
  deny  all;
}
```

Note

This example will limit access to all methods, **except** `GET` and `HEAD`.

limit_rate

Syntax	`limit_rate` *rate*;
Default	`limit_rate 0;`
Context	http, server, location, if in location

Limits the rate of response transmission to a client. The rate is specified in bytes per second. The zero value disables rate limiting. The limit is set per a request, and so if a client simultaneously opens two connections, the overall rate will be twice as much as the specified limit.

Parameter value can contain variables. It may be useful in cases where rate should be limited depending on a certain condition:

```
map $slow $rate {
  1     4k;
  2     8k;
}

limit_rate $rate;
```

Rate limit can also be set in the *$limit_rate* variable, however, this method is not recommended:

```
server {

  if ($slow) {
    set $limit_rate 4k;
  }

}
```

Rate limit can also be set in the "X-Accel-Limit-Rate" header field of a proxied server response. This capability can be disabled using the *proxy_ignore_headers*, *fastcgi_ignore_headers*, *uwsgi_ignore_headers* and *scgi_ignore_headers* directives.

limit_rate_after

Syntax	`limit_rate_after` *size*;
Default	`limit_rate_after 0;`
Context	http, server, location, if in location

Sets the initial amount after which the further transmission of a response to a client will be rate limited. Parameter value can contain variables.

Example:

```
location /flv/ {
 flv;
 limit_rate_after 500k;
 limit_rate       50k;
}
```

lingering_close

Syntax	`lingering_close on | always | off;`
Default	`lingering_close on;`
Context	http, server, location

Controls how Angie closes client connections.

`on`	instructs Angie to *wait* for and *process* additional data from a client before fully closing a connection, but only if heuristics suggests that a client may be sending more data.
`always`	will cause Angie to unconditionally wait for and process additional client data.
`off`	tells Angie to never wait for more data and close the connection immediately. This behavior breaks the protocol and should not be used under normal circumstances.

To control closing HTTP/2 connections, the directive must be specified on the *server* level.

lingering_time

Syntax	`lingering_time` *time*;
Default	`lingering_time 30s;`
Context	http, server, location

When *lingering_close* is in effect, this directive specifies the maximum time during which Angie will process (read and ignore) additional data coming from a client. After that, the connection will be closed, even if there will be more data.

lingering_timeout

Syntax	`lingering_timeout` *time*;
Default	`lingering_timeout 5s;`
Context	http, server, location

When *lingering_close* is in effect, the directive specifies the maximum waiting time for more client data to arrive. If data are not received during this time, the connection is closed. Otherwise, the data are read and ignored, and Angie starts waiting for more data again. This "wait-read-ignore" cycle is repeated no longer than specified by the *lingering_time* directive.

During a graceful shutdown, keepalive connections from clients are closed only if they remain inactive for at least the duration of `lingering_timeout`.

listen

Syntax	`listen` *address*[:*port*] [`default_server`] [`ssl`] [http2 | `quic`] [`proxy_protocol`] [`setfib=`*number*] [`fastopen=`*number*] [`backlog=`*number*] [`rcvbuf=`*size*] [`sndbuf=`*size*] [`accept_filter=`*filter*] [`deferred`] [`bind`] [`ipv6only=on` | `off`] [`reuseport`] [`so_keepalive=`on|off|[*keepidle*]:[*keepintvl*]:[*keepcnt*]]; `listen` *port* [`default_server`] [`ssl`] [http2 | `quic`] [`proxy_protocol`] [`setfib=`*number*] [`fastopen=`*number*] [`backlog=`*number*] [`rcvbuf=`*size*] [`sndbuf=`*size*] [`accept_filter=`*filter*] [`deferred`] [`bind`] [`ipv6only=on` | `off`] [`reuseport`] [`so_keepalive=`on|off|[*keepidle*]:[*keepintvl*]:[*keepcnt*]]; `listen` unix:*path* [`default_server`] [`ssl`] [http2 | `quic`] [`proxy_protocol`] [`backlog=`*number*] [`rcvbuf=`*size*] [`sndbuf=`*size*] [`accept_filter=`*filter*] [`deferred`] [`bind`] [`so_keepalive=`on|off|[*keepidle*]:[*keepintvl*]:[*keepcnt*]];
Default	`listen *:80 | *:8000;`
Context	server

Sets the `address` and `port` for listen socket, or the path for a UNIX domain socket on which the server will accept requests. An `address` may also be a hostname, for example:

```
listen 127.0.0.1:8000;
listen 127.0.0.1;
listen 8000;
listen *:8000;
listen localhost:8000;
```

IPv6 addresses are specified in square brackets:

```
listen [::]:8000;
listen [::1];
```

UNIX domain sockets are specified with the `unix:` prefix:

```
listen unix:/var/run/angie.sock;
```

Both `address` and `port`, or only `address` or `port`, can be specified. When some parts are omitted, the behavior varies:

- If only the `address` is given, port 80 is used.
- If only the `port` is given, Angie listens on all available IPv4 (and IPv6, if enabled) interfaces. The first defined `server` block for that port becomes the default for requests with an unmatched `Host` header.
- If the directive is omitted entirely, Angie uses `*:80` when running with superuser privileges or `*:8000` otherwise.

`default_server`	The server with this parameter specified will be the default server for the given `address:port` pair (together they form a *listening socket*). If there are no directives with the `default_server` parameter, the default server for the listening socket will be the first server in the configuration that serves this socket.
`ssl`	allows specifying that all connections accepted on this port should work in SSL mode. This allows for a more *compact configuration* for the server that handles both HTTP and HTTPS requests.
`http2`	configures the port to accept HTTP/2 connections. Normally, for this to work the ssl parameter should be specified as well, but Angie can also be configured to accept HTTP/2 connections without SSL. Deprecated since version 1.2.0. Use the *http2* directive instead.
`quic`	configures the port to accept QUIC connections. To use this option, Angie must have the *HTTP3 module* enabled and configured. With `quic` set, you can also specify `reuseport` so multiple worker processes can be used.
`proxy_protocol`	allows specifying that all connections accepted on this port should use the PROXY protocol.

The `listen` directive can have several additional parameters specific to socket-related system calls. These parameters can be specified in any `listen` directive, but only once for a given `address:port` pair.

`setfib=`number`	this parameter sets the associated routing table, FIB (the *SO_SETFIB* option) for the listening socket. This currently works only on FreeBSD.
`fastopen=`number	enables "TCP Fast Open" for the listening socket and limits the maximum length for the queue of connections that have not yet completed the three-way handshake.

Caution

Do not enable this feature unless the server can handle receiving the same SYN packet with data more than once.

`backlog=`number``	sets the `backlog` parameter in the `listen()` call that limits the maximum length for the queue of pending connections. By default, backlog is set to -1 on FreeBSD, DragonFly BSD, and macOS, and to 511 on other platforms.
`rcvbuf=`*size*	sets the receive buffer size (the `SO_RCVBUF` option) for the listening socket.
`sndbuf=`*size*	sets the send buffer size (the `SO_SNDBUF` option) for the listening socket.
`accept_filter=fi`	sets the name of accept filter (the `SO_ACCEPTFILTER` option) for the listening socket that filters incoming connections before passing them to `accept()`. This works only on FreeBSD and NetBSD 5.0+. Possible values are `dataready` and `httpready`.
`deferred`	instructs to use a deferred `accept()` (the `TCP_DEFER_ACCEPT` socket option) on Linux.
`bind`	instructs to make a separate `bind()` call for a given address:port pair. This is useful because if there are several *listen* directives with the same port but different addresses, and one of the `listen` directives listens on all addresses for the given `port` (`*:port`), Angie will `bind()` only to `*:port`. It should be noted that the `getsockname()` system call will be made in this case to determine the address that accepted the connection. If the `setfib`, `fastopen`, `backlog`, `rcvbuf`, `sndbuf`, `accept_filter`, `deferred`, `ipv6only`, `reuseport` or `so_keepalive` parameters are used then for a given `address:port` pair a separate `bind()` call will always be made.
`ipv6only=on` | `off`	this parameter determines (via the `IPV6_V6ONLY` socket option) whether an IPv6 socket listening on a wildcard address [::] will accept only IPv6 connections or both IPv6 and IPv4 connections. This parameter is turned on by default. It can only be set once on start.
`reuseport`	this parameter instructs to create an individual listening socket for each worker process (using the `SO_REUSEPORT` socket option on Linux 3.9+ and DragonFly BSD, or `SO_REUSEPORT_LB` on FreeBSD 12+), allowing a kernel to distribute incoming connections between worker processes. This currently works only on Linux 3.9+, DragonFly BSD, and FreeBSD 12+.

Caution

Inappropriate use of this option may have its security implications.

`so_keepalive=on` | `off` | [`keepidle`]:[*keepintvl*]:[*keepcnt*]`

Configures the "TCP keepalive" behavior for the listening socket.

`''`	if this parameter is omitted then the operating system's settings will be in effect for the socket
`on`	the `SO_KEEPALIVE` option is turned on for the socket
`off`	the `SO_KEEPALIVE` option is turned off for the socket

Some operating systems support setting of TCP keepalive parameters on a per-socket basis using the `TCP_KEEPIDLE`, `TCP_KEEPINTVL`, and `TCP_KEEPCNT` socket options. On such systems (currently, Linux 2.4+, NetBSD 5+, and FreeBSD 9.0-STABLE), they can be configured using the `keepidle`, `keepintvl`, and `keepcnt` parameters. One or two parameters may be omitted, in which case the system default setting for the corresponding socket option will be in effect. For example,

```
so_keepalive=30m::10
```

will set the idle timeout (`TCP_KEEPIDLE`) to 30 minutes, leave the probe interval (`TCP_KEEPINTVL`) at its system default, and set the probes count (`TCP_KEEPCNT`) to 10 probes.

Example:

```
listen 127.0.0.1 default_server accept_filter=dataready backlog=1024;
```

location

Syntax	`location` ([= | ~ | ~* | ^~] *uri* | *@name*)+ { ... }
Default	—
Context	server, location

Sets the configuration depending on whether the request URI matches any of the matching expressions.

The matching is performed against a normalized URI, after decoding the text encoded in the "%XX" form, resolving references to relative path components "." and "..", and possible *compression* of two or more adjacent slashes into a single slash.

A `location` can either be defined by a prefix string, or by a regular expression.

Regular expressions are specified with the preceding modifier:

`~*`	Case-insensitive matching
`~`	Case-sensitive matching

To find a location that matches a request, Angie first checks the locations defined with prefix strings (known as prefix locations). Among them, the location with the longest matching prefix is selected and tentatively stored.

> **Note**
>
> For case-insensitive operating systems such as macOS, prefix string matching is case insensitive. However, matching is limited to single-byte locales.

Then, regex-based locations are evaluated in order of their appearance in the configuration file. Their evaluation stops at the first match, and the corresponding configuration is used. If no matching regex location is found, Angie uses the configuration of the tentatively stored prefix location.

With some exceptions mentioned below, `location` blocks can be nested.

Regex locations may define capture groups that can later be used with other directives.

If the matching prefix location uses the `^~` modifier, regex locations aren't checked.

Also, the `=` modifier enables exact URI matching mode for a `location`; if an exact match is found, the lookup stops. For example, if `/` requests are frequent, defining `location =/` speeds up their processing because the lookup stops at the exact match. Obviously, such locations can't contain nested locations.

Example:

```
location =/ {
    #configuration A
}

location / {
    #configuration B
}

location /documents/ {
    #configuration C
}
```

```
location ^~/images/ {
   #configuration D
}

location ~*\.(gif|jpg|jpeg)$ {
   #configuration E
}
```

- A / request matches configuration A,
- an /index.html request matches configuration B,
- a /documents/document.html request matches configuration C,
- an /images/1.gif request matches configuration D,
- and a /documents/1.jpg request matches configuration E.

Note

If a prefix location ends with a slash character and *auto_redirect* is enabled, the following occurs: When a request arrives with the URI that has no trailing slash but otherwise matches the prefix exactly, a permanent 301 code redirect is returned, pointing to the requested URI with the slash appended.

With an exact URI-matching location, redirection isn't applied:

```
location /user/ {
  proxy_pass http://user.example.com;
}

location =/user {
  proxy_pass http://login.example.com;
}
```

The @ prefix defines a *named* location. Such locations aren't used for regular request processing, but instead can be used for request redirection. They cannot be nested and cannot contain nested locations.

Combined locations

Several location contexts that define identical configuration blocks can be compacted by listing all their matching expressions in a single location with a single configuration block. That's called a *combined* location.

Suppose that configurations A, D, and E from the previous example define identical configurations; you can combine them into one location:

```
location =/
         ^~/images/
         ~*\.(gif|jpg|jpeg)$ {
   # general configuration
}
```

A named location can also be a part of the combination:

```
location =/
         @named_combined {
   #...
}
```

Caution

A combined `location` can't have a space between the matching expression and its modifier. Proper form: `location ~*/match(ing|es|er)$ ....`

Note

Currently, a combined `location` cannot **immediately** contain neither *proxy_pass* and similar directives with URI set, nor `api` or `alias`. However, these directives can be used by locations nested inside a combined location.

log_not_found

Syntax	`log_not_found on | off;`
Default	`log_not_found on;`
Context	http, server, location

Enables or disables logging of errors about not found files into *error_log*.

log_subrequest

Syntax	`log_subrequest on | off;`
Default	`log_subrequest off;`
Context	http, server, location

Enables or disables logging of subrequests into *access_log*.

max_headers

Syntax	`max_headers` *number*;
Default	`max_headers 1000;`
Context	http, server

Sets the maximum number of client request header fields allowed. If this limit is exceeded, a `400 (Bad Request)` error is returned.

When this directive is set at the *server* level, the value from the default server may be applied. For more information, refer to the *Virtual server selection* section.

max_ranges

Syntax	`max_ranges` *number*;
Default	—
Context	http, server, location

Limits the maximum allowed number of ranges in byte-range requests. Requests that exceed the limit are processed as if there were no byte ranges specified. By default, the number of ranges is not limited.

`0`	disables the byte-range support completely

merge_slashes

Syntax	`merge_slashes on | off;`
Default	`merge_slashes on;`
Context	http, server

Enables or disables compression of two or more adjacent slashes in a URI into a single slash.

Note that compression is essential for the correct matching of prefix string and regular expression locations. Without it, the `//scripts/one.php` request would not match

```
location /scripts/ { }
```

and might be processed as a static file. So it gets converted to `/scripts/one.php`.

Turning the compression off can become necessary if a URI contains base64-encoded names, since `base64` uses the "/" character internally. However, for security considerations, it is better to avoid turning the compression off.

If the directive is specified on the *server* level, the value from the default server can be used.

msie_padding

Syntax	`msie_padding on | off;`
Default	`msie_padding on;`
Context	http, server, location

Enables or disables adding comments to responses for MSIE clients with status greater than 400 to increase the response size to 512 bytes.

msie_refresh

Syntax	`msie_refresh on | off;`
Default	`msie_refresh off;`
Context	http, server, location

Enables or disables issuing refreshes instead of redirects for MSIE clients.

open_file_cache

Syntax	`open_file_cache off;` `open_file_cache max=`*N* `[inactive=`*time*`];`
Default	`open_file_cache off;`
Context	http, server, location

Configures a cache that can store:

- open file descriptors, their sizes and modification times;
- information on existence of directories;
- file lookup errors, such as "file not found", "no read permission", and so on.

Caching of errors should be enabled separately by the *open_file_cache_errors* directive.

`max`	sets the maximum number of elements in the cache; on cache overflow the least recently used (LRU) elements are removed;
`inactive`	defines a time after which an element is removed from the cache if it has not been accessed during this time; By default, it is set to 60 seconds.
`off`	disables the cache.

Example:

```
open_file_cache          max=1000 inactive=20s;
open_file_cache_valid    30s;
open_file_cache_min_uses 2;
open_file_cache_errors   on;
```

open_file_cache_errors

Syntax	`open_file_cache_errors on	off;`
Default	`open_file_cache_errors off;`
Context	http, server, location

Enables or disables caching of file lookup errors by *open_file_cache*.

open_file_cache_min_uses

Syntax	`open_file_cache_min_uses` *number*;
Default	`open_file_cache_min_uses 1;`
Context	http, server, location

Sets the minimum`number` of file accesses during the period configured by the `inactive` parameter of the *open_file_cache* directive, required for a file descriptor to remain open in the cache.

open_file_cache_valid

Syntax	`open_file_cache_valid` *time*;
Default	`open_file_cache_valid 60s;`
Context	http, server, location

Sets a time after which *open_file_cache* elements should be validated.

output_buffers

Syntax	`output_buffers` *number size*;
Default	`output_buffers 2 32k;`
Context	http, server, location

Sets the`number` and *size* of the buffers used for reading a response from a disk.

port_in_redirect

Syntax	`port_in_redirect on | off;`
Default	`port_in_redirect on;`
Context	http, server, location

Enables or disables specifying the port in *absolute* redirects issued by Angie.

The use of the primary server name in redirects is controlled by the *server_name_in_redirect* directive.

postpone_output

Syntax	`postpone_output` *size*;
Default	`postpone_output 1460;`
Context	http, server, location

If possible, the transmission of client data will be postponed until Angie has at least *size* bytes of data to send.

`0`	disables postponing data transmission

read_ahead

Syntax	`read_ahead` *size*;
Default	`read_ahead 0;`
Context	http, server, location

Sets the amount of pre-reading for the kernel when working with file.

On Linux, the `posix_fadvise(0, 0, 0, POSIX_FADV_SEQUENTIAL)` system call is used, and so the size parameter is ignored.

recursive_error_pages

Syntax	`recursive_error_pages on | off;`
Default	`recursive_error_pages off;`
Context	http, server, location

Enables or disables doing several redirects using the *error_page* directive. The number of such redirects is *limited*.

request_pool_size

Syntax	`request_pool_size` *size*;
Default	`request_pool_size 4k;`
Context	http, server

Allows accurate tuning of per-request memory allocations. This directive has minimal impact on performance and should not generally be used.

reset_timedout_connection

Syntax	`reset_timedout_connection on | off;`
Default	`reset_timedout_connection off;`
Context	http, server, location

Enables or disables resetting timed out connections and connections closed with the non-standard code 444. The reset is performed as follows. Before closing a socket, the `SO_LINGER` option is set on it with a timeout value of `0`. When the socket is closed, `TCP RST` is sent to the client, and all memory occupied by this socket is released. This helps avoid keeping an already closed socket with filled buffers in a `FIN_WAIT1` state for a long time.

Note

timed out keep-alive connections are closed normally.

resolver

Syntax	`resolver` *address* ... [`valid=`*time*] [`ipv4=on` | `off`] [`ipv6=on` | `off`] [`status_zone=`*zone*];
Default	—
Context	http, server, location, upstream

Configures name servers used to resolve names of upstream servers into addresses, for example:

```
resolver 127.0.0.53 [::1]:5353;
```

The address can be specified as a domain name or IP address, with an optional port. If port is not specified, the port 53 is used. Name servers are queried in a round-robin fashion.

By default, Angie caches answers using the TTL value of a response.

`valid`	*optional* parameter allows overriding cached entry validity

```
resolver 127.0.0.53 [::1]:5353 valid=30s;
```

By default, Angie will look up both IPv4 and IPv6 addresses while resolving.

`ipv4=off`	disables looking up of IPv4 addresses
`ipv6=off`	disables looking up of IPv6 addresses

`status_zone`	*optional* parameter; enables the collection of DNS server request and response metrics (*/status/resolvers/<zone>*) in the specified zone.

Tip

To prevent DNS spoofing, it is recommended configuring DNS servers in a properly secured trusted local network.

Tip

When running in Docker, use its internal DNS server address such as `127.0.0.11`.

resolver_timeout

Syntax	`resolver_timeout` *time*;
Default	`resolver_timeout 30s;`
Context	http, server, location, upstream

Sets a timeout for name resolution, for example:

```
resolver_timeout 5s;
```

root

Syntax	`root` *path*;
Default	`root html;`
Context	http, server, location, if in location

Sets the root directory for requests. For example, with the following configuration

```
location /i/ {
  root /data/w3;
}
```

The `/data/w3/i/top.gif` file will be sent in response to the `/i/top.gif` request.

The `path` value can contain variables, except *$document_root* and *$realpath_root*.

A path to the file is constructed by merely adding a URI to the value of the root directive. If a URI has to be modified, the *alias* directive should be used.

satisfy

Syntax	`satisfy all` | `any`;
Default	`satisfy all;`
Context	http, server, location

Allows access if all (`all`) or at least one (`any`) of these modules allow access: *Access*, *Auth Basic*, or *Auth Request*.

```
location / {
  satisfy any;

  allow 192.168.1.0/32;
  deny  all;

  auth_basic           "closed site";
  auth_basic_user_file conf/htpasswd;
}
```

send_lowat

Syntax	`send_lowat` *size*;
Default	`send_lowat 0;`
Context	http, server, location

If the directive is set to a non-zero value, Angie will try to minimize the number of send operations on client sockets by using either `NOTE_LOWAT` flag of the ref:*kqueue* method or the `SO_SNDLOWAT` socket option. In both cases the specified size is used.

send_timeout

Syntax	`send_timeout` *time*;
Default	`send_timeout 60s;`
Context	http, server, location

Sets a timeout for transmitting a response to the client. The timeout is set only between two successive write operations, not for the transmission of the whole response. If the client does not receive anything within this time, the connection is closed.

sendfile

Syntax	`sendfile on` | `off`;
Default	`sendfile off;`
Context	http, server, location, if in location

Enables or disables the use of `sendfile()`.

aio can be used to pre-load data for `sendfile()`:

```
location /video/ {
  sendfile       on;
  tcp_nopush     on;
  aio            on;
}
```

In this configuration, `sendfile()` is called with the `SF_NODISKIO` flag which causes it not to block on disk I/O, but, instead, report back that the data are not in memory. Angie then initiates an asynchronous data load by reading one byte. On the first read, the FreeBSD kernel loads the first 128K bytes of a file into memory, although next reads will only load data in 16K chunks. This can be changed using the *read_ahead* directive.

sendfile_max_chunk

Syntax	`sendfile_max_chunk` *size*;
Default	`sendfile_max_chunk 2m;`
Context	http, server, location

Limits the amount of data that can be transferred in a single `sendfile()` call. Without the limit, one fast connection may seize the worker process entirely.

server

Syntax	`server { ... }`
Default	—
Context	http

Sets configuration for a virtual server. There is no clear separation between IP-based (based on the IP address) and name-based (based on the "Host" request header field) virtual servers. Instead, the *listen* directives describe all addresses and ports that should accept connections for the server, and the *server_name* directive lists all server names.

Example configurations are provided in the *How Angie processes a request* document.

server_name

Syntax	`server_name` *name* ...;
Default	`server_name "";`
Context	server

Sets names of a virtual server, for example:

```
server {
  server_name example.com www.example.com;
}
```

The first name becomes the primary server name.

Server names can include an asterisk ("*") replacing the first or last part of a name:

```
server {
  server_name example.com *.example.com www.example.*;
}
```

Such names are called wildcard names.

The first two of the names mentioned above can be combined in one:

```
server {
  server_name .example.com;
}
```

It is also possible to use regular expressions in server names, preceding the name with a tilde ("~"):

```
server {
  server_name ~^www\d+\.example\.com$ www.example.com;
}
```

Regular expressions can contain captures that can later be used in other directives:

```
server {
  server_name ~^(www\.)?(.+)$;

  location / {
     root /sites/$2;
  }
}

server {
```

```
  server_name _;

  location / {
     root /sites/default;
  }
}
```

Named captures in regular expressions create variables that can later be used in other directives:

```
  server {
    server_name ~^(www\.)?(?<domain>.+)$;

    location / {
       root /sites/$domain;
    }
  }

server {
    server_name _;

    location / {
       root /sites/default;
    }
  }
```

Note

If the directive is set to *$hostname*, the hostname of the web server is used.

You can also specify an empty server name (`""`):

```
server {
    server_name www.example.com "";
}
```

When searching for a virtual server by a name that is matched by multiple options (for example, both a wildcard and a regular expression), the first matching option will be selected in the following priority order:

- exact name;
- longest name with a wildcard at the beginning, such as `*.example.com`;
- longest name with a wildcard at the end, such as `mail.*`;
- the first matching regular expression (in the order of appearance), including an empty name.

Attention

To make `server_name` work with TLS, you need to terminate the TLS connection. The directive matches the `Host` in an HTTP request, so the handshake must be completed and the connection decrypted.

server_name_in_redirect

Syntax	`server_name_in_redirect on | off;`
Default	`server_name_in_redirect off;`
Context	http, server, location

Enables or disables the use of the primary server name, specified by the *server_name* directive, in *absolute redirects* issued by Angie.

`on`	the primary server name, specified by the *server_name* directive
`off`	the name from the "Host" request header field is used. If this field is not present, the IP address of the server is used.

The use of a port in redirects is controlled by the *port_in_redirect* directive.

server_names_hash_bucket_size

Syntax	`server_names_hash_bucket_size` *size*;
Default	`server_names_hash_bucket_size 32 | 64 | 128;`
Context	http

Sets the bucket size for the server names hash tables. The default value depends on the size of the processor's cache line. The details of setting up hash tables are provided in a separate *document*.

server_names_hash_max_size

Syntax	`server_names_hash_max_size` *size*;
Default	`server_names_hash_max_size 512;`
Context	http

Sets the maximum size of the server names hash tables. The details of setting up hash tables are provided in a separate *document*.

server_tokens

Syntax	`server_tokens on | off | build |` *string*;
Default	`server_tokens` *on*;
Context	http, server, location

Enables or disables emitting Angie version on error pages and in the `Server` response header field. The `build` parameter enables emitting the build name, set by the respective configure parameter, along with the version.

Added in version 1.1.0: PRO

In Angie PRO, if the directive sets a `string`, which may also contain variables, the error pages and the `Server` response header field will use the string's variable-interpolated value instead of server name, version, and build name. An empty `string` disables emitting the `Server` field.

status_zone

Syntax	`status_zone off` | *zone* | *key* `zone=`*zone*[:*count*];
Default	—
Context	server, location, if in location

Allocates a shared memory zone to collect metrics for */status/http/location_zones/<zone>* and */status/http/server_zones/<zone>*.

Multiple `server` contexts can share the same zone for data collection; the special value `off` disables data collection in nested `location` blocks.

The single-value `zone` syntax aggregates all metrics for its context in the same shared memory zone:

```
server {

    listen 80;
    server_name *.example.com;

    status_zone single;
    # ...
}
```

The alternative syntax uses the folowing parameters:

`key`	A string with variables, whose value determines the grouping of requests in the zone. All requests producing identical values after substitution are grouped together. If substitution yields an empty value, metrics aren't updated.
`zone`	The name of the shared memory zone.
`count` (optional)	The maximum number of separate groups for collecting metrics. If new `key` values would exceed this limit, they are grouped under `zone` instead. The default value is 1.

In the following example, all requests sharing the same `$host` value are grouped into the `host_zone`. Metrics are tracked separately for each unique `$host` until there are 10 metric groups. Once this limit is reached, any additional `$host` values are included under the `host_zone`:

```
server {

    listen 80;
    server_name *.example.com;

    status_zone $host zone=host_zone:10;

    location / {

        proxy_pass http://example.com;
    }
}
```

The resulting metrics are thus split between individual hosts in the API output.

subrequest_output_buffer_size

Syntax	`subrequest_output_buffer_size` *size*;
Default	`subrequest_output_buffer_size 4k` | `8k`;
Context	http, server, location

Sets the size of the buffer used for storing the response body of a subrequest. By default, the buffer size is equal to one memory page. This is either `4K` or `8K`, depending on a platform. It can be made smaller, however.

> **Note**
>
> The directive is applicable only for subrequests with response bodies saved into memory. For example, such subrequests are created by *SSI*.

tcp_nodelay

Syntax	`tcp_nodelay on` | `off`;
Default	`tcp_nodelay on`;
Context	http, server, location

Enables or disables the use of the `TCP_NODELAY` option. The option is enabled when a connection is transitioned into the keep-alive state. Additionally, it is enabled on SSL connections, for unbuffered proxying, and for *WebSocket proxying*.

tcp_nopush

Syntax	`tcp_nopush on` | `off`;
Default	`tcp_nopush off`;
Context	http, server, location

Enables or disables the use of the `TCP_NOPUSH` socket option on FreeBSD or the `TCP_CORK` socket option on Linux. The options are enabled only when *sendfile* is used. Enabling the option allows

- sending the response header and the beginning of a file in one packet, on Linux and FreeBSD 4.*;
- sending a file in full packets.

try_files

Syntax	`try_files` file ... *uri*; `try_files` file ... =*code*;
Default	—
Context	server, location

Checks the existence of files in the specified order and uses the first found file for request processing; the processing is performed in the current context. The path to a file is constructed from the file parameter according to the *root* and *alias* directives. It is possible to check directory's existence by specifying a slash at the end of a name, e.g. `$uri/`. If none of the files were found, an internal redirect to the `uri` specified in the last parameter is made. For example:

```
location /images/ {
  try_files $uri /images/default.gif;
}

location = /images/default.gif {
  expires 30s;
}
```

The last parameter can also point to a named `location`, as shown in examples below. The last parameter can also be a code:

```
location / {
  try_files $uri $uri/index.html $uri.html =404;
}
```

In the following example,

```
location / {
  try_files $uri $uri/ @drupal;
}
```

the `try_files` directive is equivalent to

```
location / {
  error_page 404 = @drupal;
  log_not_found off;
}
```

And here,

```
location ~ \.php$ {
  try_files $uri @drupal;

  fastcgi_pass ...;

  fastcgi_param SCRIPT_FILENAME /path/to$fastcgi_script_name;

#  ...
}
```

`try_files` checks the existence of the PHP file before passing the request to the FastCGI server.

```
location / {
  try_files /system/maintenance.html
            $uri $uri/index.html $uri.html
            @mongrel;
}

location @mongrel {
  proxy_pass http://mongrel;
}
```

```
location / {
  try_files $uri $uri/ @drupal;
}

location ~ \.php$ {
  try_files $uri @drupal;

  fastcgi_pass ...;

  fastcgi_param SCRIPT_FILENAME /path/to$fastcgi_script_name;
  fastcgi_param SCRIPT_NAME     $fastcgi_script_name;
  fastcgi_param QUERY_STRING    $args;

#  ... other fastcgi_param
}
```

```
location @drupal {
  fastcgi_pass ...;

  fastcgi_param SCRIPT_FILENAME /path/to/index.php;
  fastcgi_param SCRIPT_NAME     /index.php;
  fastcgi_param QUERY_STRING    q=$uri&$args;

#  ... other fastcgi_param
}
```

```
location / {
  try_files $uri $uri/ @wordpress;
}

location ~ \.php$ {
  try_files $uri @wordpress;

  fastcgi_pass ...;

  fastcgi_param SCRIPT_FILENAME /path/to$fastcgi_script_name;
#  ... other fastcgi_param
}

location @wordpress {
  fastcgi_pass ...;

  fastcgi_param SCRIPT_FILENAME /path/to/index.php;
#  ... other fastcgi_param
}
```

types

Syntax	`types { ... }`
Default	`types` *`text/html html; image/gif gif; image/jpeg jpg;`*
Context	http, server, location

Maps file name extensions to MIME types of responses. Extensions are case-insensitive. Several extensions can be mapped to one type, for example:

```
types {
  application/octet-stream bin exe dll;
  application/octet-stream deb;
  application/octet-stream dmg;
}
```

A sufficiently full mapping table is distributed with Angie in the `conf/mime.types` file.

To make a particular location emit the "application/octet-stream" MIME type for all requests, the following configuration can be used:

```
location /download/ {
  types        { }
  default_type application/octet-stream;
}
```

types_hash_bucket_size

Syntax	`types_hash_bucket_size` *size*;
Default	`types_hash_bucket_size 64;`
Context	http, server, location

Sets the bucket size for the types hash tables. The details of setting up hash tables are provided in a separate *document*.

types_hash_max_size

Syntax	`types_hash_max_size` *size*;
Default	`types_hash_max_size 1024;`
Context	http, server, location

Sets the maximum size of the types hash tables. The details of setting up hash tables are provided in a separate *document*.

underscores_in_headers

Syntax	`underscores_in_headers on | off;`
Default	`underscores_in_headers off;`
Context	http, server

Enables or disables the use of underscores in client request header fields. When the use of underscores is disabled, request header fields whose names contain underscores are marked as invalid and become subject to the *ignore_invalid_headers* directive.

If the directive is specified on the *server* level, the value from the default server can be used.

variables_hash_bucket_size

Syntax	`variables_hash_bucket_size` *size*;
Default	`variables_hash_bucket_size 64;`
Context	http

Sets the bucket size for the variables hash table. The details of setting up hash tables are provided in a separate *document*.

variables_hash_max_size

Syntax	`variables_hash_max_size` *size*;
Default	`variables_hash_max_size 1024;`
Context	http

Sets the maximum size of the variables hash table. The details of setting up hash tables are provided in a separate *document*.

Built-in Variables

The `http_core` module supports built-in variables with names matching the Apache Server variables. First of all, these are variables representing client request header fields, such as `$http_user_agent`, `$http_cookie`, and so on. Also, there are other variables:

`$angie_version`

Angie version

`$arg_<name>`

argument with the specified *name* in the request line

`$args`

arguments in the request line

`$binary_remote_addr`

client address in a binary form, value's length is always 4 bytes for IPv4 addresses or 16 bytes for IPv6 addresses

`$body_bytes_sent`

number of bytes sent to the client, not counting the response header; this variable is compatible with the "%B" parameter of the `mod_log_config` Apache module

`$bytes_sent`

number of bytes sent to a client

`$connection`

connection serial number

`$connection_requests`

current number of requests made through a connection

`$connection_time`

connection time in seconds with a milliseconds resolution

`$content_length`

"Content-Length" request header field

`$content_type`

"Content-Type" request header field

`$cookie_<name>`

cookie with the specified *name*

`$document_root`

root or *alias* directive's value for the current request

`$document_uri`

same as *$uri*

`$host`

in this order of precedence: host name from the request line, or host name from the "Host" request header field, or the server name matching a request

`$hostname`

host name

`$http_<name>`

arbitrary request header field; the *name* is the field name converted to lower case with dashes replaced by underscores

`$https`

`on` if connection operates in SSL mode, or an empty string otherwise

`$is_args`

`?`, if a request line has arguments, or an empty string otherwise

`$limit_rate`

setting this variable enables response rate limiting; see *limit_ rate*

`$msec`

current time in seconds with the milliseconds resolution

`$pid`

PID of the worker process

`$pipe`

`p` if request was pipelined, `.` otherwise

`$proxy_protocol_addr`

Client address from the PROXY protocol header.

The PROXY protocol must be previously enabled by setting the `proxy_protocol` parameter in the *listen* directive.

`$proxy_protocol_port`

Client port from the PROXY protocol header.

The PROXY protocol must be previously enabled by setting the `proxy_protocol` parameter in the *listen* directive.

`$proxy_protocol_server_addr`

Server address from the PROXY protocol header.

The PROXY protocol must be previously enabled by setting the `proxy_protocol` parameter in the *listen* directive.

`$proxy_protocol_server_port`

Server port from the PROXY protocol header.

The PROXY protocol must be previously enabled by setting the `proxy_protocol` parameter in the *listen* directive.

`$proxy_protocol_tlv_<name>`

TLV from the PROXY Protocol header. The *name* can be a TLV type or its numeric value. In the latter case, the value is hexadecimal and should be prefixed with `0x`:

```
$proxy_protocol_tlv_alpn
$proxy_protocol_tlv_0x01
```

SSL TLVs can also be accessed by TLV type name or its numeric value, both prefixed by `ssl_`:

```
$proxy_protocol_tlv_ssl_version
$proxy_protocol_tlv_ssl_0x21
```

The following TLV type names are supported:

- `alpn (0x01)` - upper layer protocol used over the connection
- `authority (0x02)` - host name value passed by the client
- `unique_id (0x05)` - unique connection id
- `netns (0x30)` - name of the namespace
- `ssl (0x20)` - binary SSL TLV structure

The following SSL TLV type names are supported:

- `ssl_version (0x21)` - SSL version used in client connection
- `ssl_cn (0x22)` - SSL certificate Common Name
- `ssl_cipher (0x23)` - name of the used cipher
- `ssl_sig_alg (0x24)` - algorithm used to sign the certificate
- `ssl_key_alg (0x25)` - public-key algorithm

Also, the following special SSL TLV type name is supported:

- `ssl_verify` - client SSL certificate verification result, `0` if the client presented a certificate and it was successfully verified, non-zero otherwise.

The PROXY protocol must be previously enabled by setting the `proxy_protocol` parameter in the *listen* directive.

`$query_string`

same as *$args*

`$realpath_root`

an absolute pathname corresponding to the *root* or *alias* directive's value for the current request, with all symbolic links resolved to real paths

`$remote_addr`

client address

`$remote_port`

client port

`$remote_user`

user name supplied with the Basic authentication

`$request`

full original request line

`$request_body`

Request body.

The variable's value is *made available* in locations processed by the *proxy_pass*, *fastcgi_pass*, *uwsgi_pass* and *scgi_pass* directives when the request body was read to a *memory buffer*.

`$request_body_file`

Name of a temporary file with the request body.

At the end of processing, the file needs to be removed. To always write the request body to a file, *client_body_in_file_only* needs to be enabled. When the name of a temporary file is passed in a proxied request or in a request to a FastCGI/uwsgi/SCGI server, passing the request body should be disabled by the *proxy_pass_request_body off*, *fastcgi_pass_request_body off*, *uwsgi_pass_request_body off* or *scgi_pass_request_body off* directives, respectively.

`$request_completion`

"OK" if a request has completed, or an empty string otherwise

`$request_filename`

file path for the current request, based on the *root* or *alias* directives, and the request URI

`$request_id`

unique request identifier generated from 16 random bytes, in hexadecimal

`$request_length`

request length (including request line, header, and request body)

`$request_method`

request method, usually `GET` or `POST`

`$request_time`

request processing time in seconds with a milliseconds resolution; time elapsed since the first bytes were read from the client

`$request_uri`

full original request URI (with arguments)

`$scheme`

request scheme, "http" or "https"

`$sent_http_<name>`

arbitrary response header field; the *name* is the field name converted to lower case with dashes replaced by underscores

`$sent_trailer_<name>`

arbitrary field sent at the end of the response; the *name* is the field name converted to lower case with dashes replaced by underscores

`$server_addr`

The address of the server which accepted a request. Computing the variable's value usually requires one system call. To avoid it, the *listen* directives must specify addresses and use the `bind` parameter.

`$server_name`

name of the server which accepted a request

`$server_port`

port of the server which accepted a request

`$server_protocol`

request protocol, usually "HTTP/1.0", "HTTP/1.1", or "HTTP/2.0"

`$status`

response status

`$time_iso8601`

local time in the ISO 8601 standard format

`$time_local`

local time in the Common Log Format

`$tcpinfo_rtt, $tcpinfo_rttvar, $tcpinfo_snd_cwnd, $tcpinfo_rcv_space`

information about the client TCP connection; available on systems that support the `TCP_INFO` socket option

`$uri`

Current URI in the request, *normalized*. The value of `$uri` may change during request processing, e.g. when doing internal redirects, or when using index files

Stream Module

Access

The module allows limiting access to certain client addresses.

Configuration Example

```
server {
    ...
    deny  192.168.1.1;
    allow 192.168.1.0/24;
    allow 10.1.1.0/16;
    allow 2001:0db8::/32;
    deny  all;
}
```

The rules are checked in sequence until the first match is found. In this example, access is allowed only for IPv4 networks `10.1.1.0/16` and `192.168.1.0/24` excluding the address `192.168.1.1`, and for IPv6 network `2001:0db8::/32`.

Directives

allow

Syntax	`allow` *address* | *CIDR* | `unix:` | `all`;
Default	—
Context	http, server, location, limit_except

Allows access for the specified network or address. If the special value `unix:` is specified (1.5.1), allows access for all UNIX domain sockets.

deny

Syntax	`deny` *address* | *CIDR* | `unix:` | `all`;
Default	—
Context	http, server, location, limit_except

Denies access for the specified network or address. If the special value `unix:` is specified (1.5.1), denies access for all UNIX domain sockets.

Geo

The module creates variables with values depending on the client IP address.

Configuration Example

```
geo $geo {
    default        0;
```

```
    127.0.0.1       2;
    192.168.1.0/24  1;
    10.1.0.0/16     1;

    ::1             2;
    2001:0db8::/32  1;
}
```

Directives

geo

Syntax	geo [*$address*] *$variable* { ... }
Default	—
Context	stream

Describes the dependency of values of the specified variable on the client IP address. By default, the address is taken from the *$remote_addr* variable, but it can also be taken from another variable, for example:

```
geo $arg_remote_addr $geo {
    ...;
}
```

Note

Since variables are evaluated only when used, the mere existence of even a large number of declared `geo` variables does not cause any extra costs for connection processing.

If the value of a variable does not represent a valid IP address then the "255.255.255.255" address is used.

Addresses are specified either as prefixes in CIDR notation (including individual addresses) or as ranges.

The following special parameters are also supported:

`delete`	deletes the specified network
`default`	the value set to the variable if the client address does not match any of the specified addresses. When addresses are specified in CIDR notation, "`0.0.0.0/0`" and "`::/0`" can be used instead of `default`. When `default` is not specified, the default value will be an empty string
`include`	includes a file with addresses and values. There can be several inclusions.
`ranges`	indicates that addresses are specified as ranges. This parameter should be the first. To speed up loading of a geo base, addresses should be put in ascending order.

Example:

```
geo $country {
    default         ZZ;
    include         conf/geo.conf;
    delete          127.0.0.0/16;
    proxy           192.168.100.0/24;
    proxy           2001:0db8::/32;
```

```
    127.0.0.0/24   US;
    127.0.0.1/32   RU;
    10.1.0.0/16    RU;
    192.168.1.0/24 UK;
}
```

The `conf/geo.conf` file could contain the following lines:

```
10.2.0.0/16    RU;
192.168.2.0/24 RU;
```

A value of the most specific match is used. For example, for the `127.0.0.1` address the value `RU` will be chosen, not `US`.

Example with ranges:

```
geo $country {
    ranges;
    default                   ZZ;
    127.0.0.0-127.0.0.0       US;
    127.0.0.1-127.0.0.1       RU;
    127.0.0.2-127.0.0.255     US;
    10.1.0.0-10.1.255.255     RU;
    192.168.1.0-192.168.1.255 UK;
}
```

GeoIP

Creates variables with values depending on the client IP address, using the precompiled MaxMind databases.

When using the databases with IPv6 support, IPv4 addresses are looked up as IPv4-mapped IPv6 addresses.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-stream_geoip_module` build option.

Important

This module requires the MaxMind GeoIP library.

Configuration Example

```
stream {
    geoip_country         GeoIP.dat;
    geoip_city            GeoLiteCity.dat;

    map $geoip_city_continent_code $nearest_server {
        default        example.com;
        EU          eu.example.com;
        NA          na.example.com;
        AS          as.example.com;
    }
#   ...
}
```

Directives

geoip_country

Syntax	`geoip_country` *file*;
Default	—
Context	stream

Specifies a database used to determine the country depending on the client IP address. The following variables are available when using this database:

`$geoip_country_c`	two-letter country code, for example, "RU", "US".
`$geoip_country_c`	three-letter country code, for example, "RUS", "USA".
`$geoip_country_n`	country name, for example, "Russian Federation", "United States".

geoip_city

Syntax	`geoip_city` *file*;
Default	—
Context	stream

Specifies a database used to determine the country, region, and city depending on the client IP address. The following variables are available when using this database:

`$geoip_city_cont`	two-letter continent code, for example, "EU", "NA".
`$geoip_city_coun`	two-letter country code, for example, "RU", "US".
`$geoip_city_coun`	three-letter country code, for example, "RUS", "USA".
`$geoip_city_coun`	country name, for example, "Russian Federation", "United States".
`$geoip_dma_code`	DMA region code in US (also known as "metro code"), according to the geotargeting in Google AdWords API.
`$geoip_latitude`	latitude.
`$geoip_longitude`	longitude.
`$geoip_region`	two-symbol country region code (region, territory, state, province, federal land and the like), for example, "48", "DC".
`$geoip_region_na`	country region name (region, territory, state, province, federal land and the like), for example, "Moscow City", "District of Columbia".
`$geoip_city`	city name, for example, "Moscow", "Washington".
`$geoip_postal_co`	postal code.

geoip_org

Syntax	`geoip_org` *file*;
Default	—
Context	stream

Specifies a database used to determine the organization depending on the client IP address. The following variable is available when using this database:

`$geoip_org`	organization name, for example, "The University of Melbourne".

JS

The module is used to implement handlers in njs — a subset of the JavaScript language.

In our repositories, the module is built dynamically and is available as a separate package named `angie-module-njs` or `angie-pro-module-njs`.

Configuration Example

```
stream {
    js_import stream.js;

    js_set $bar stream.bar;
    js_set $req_line stream.req_line;

    server {
        listen 12345;

        js_preread stream.preread;
        return     $req_line;
    }

    server {
        listen 12346;

        js_access  stream.access;
        proxy_pass 127.0.0.1:8000;
        js_filter  stream.header_inject;
    }
}

http {
    server {
        listen 8000;
        location / {
            return 200 $http_foo\n;
        }
    }
}
```

The `stream.js` file:

```
var line = '';

function bar(s) {
    var v = s.variables;
    s.log("hello from bar() handler!");
    return "bar-var" + v.remote_port + "; pid=" + v.pid;
}

function preread(s) {
    s.on('upload', function (data, flags) {
        var n = data.indexOf('\n');
        if (n != -1) {
            line = data.substr(0, n);
            s.done();
        }
    });
}
```

```
function req_line(s) {
    return line;
}

// Read HTTP request line.
// Collect bytes in 'req' until
// request line is read.
// Injects HTTP header into a client's request

var my_header =  'Foo: foo';
function header_inject(s) {
    var req = '';
    s.on('upload', function(data, flags) {
        req += data;
        var n = req.search('\n');
        if (n != -1) {
            var rest = req.substr(n + 1);
            req = req.substr(0, n + 1);
            s.send(req + my_header + '\r\n' + rest, flags);
            s.off('upload');
        }
    });
}

function access(s) {
    if (s.remoteAddress.match('^192.*')) {
        s.deny();
        return;
    }

    s.allow();
}

export default {bar, preread, req_line, header_inject, access};
```

Directives

js_access

Syntax	`js_access` *function* | *module.function*;
Default	—
Context	stream, server

Sets an njs function which will be called at the *access phase*. Module functions can be referenced.

The function is called once at the moment when the stream session reaches the *access phase* for the first time. The function is called with the following arguments:

`s`	the stream session object

At this phase, it is possible to perform initialization or register a callback with the s.on() method for each incoming data chunk until one of the following methods are called: s.done(), s.decline(), s.allow(). As soon as one of these methods is called, the stream session processing switches to the *next phase* and all current s.on() callbacks are dropped.

js_fetch_buffer_size

Syntax	`js_fetch_buffer_size` *size*;
Default	`js_fetch_buffer_size 16k;`
Context	stream, server

Sets the size of the buffer used for reading and writing with Fetch API.

js_fetch_ciphers

Syntax	`js_fetch_ciphers` *ciphers*;
Default	`js_fetch_ciphers HIGH:!aNULL:!MD5;`
Context	stream, server

Specifies the enabled ciphers for HTTPS connections with Fetch API. The ciphers are specified in the format understood by the OpenSSL library.

The list of ciphers depends on the version of OpenSSL installed. The full list can be viewed using the `openssl ciphers` command.

js_fetch_max_response_buffer_size

Syntax	`js_fetch_max_response_buffer_size` *size*;
Default	`js_fetch_max_response_buffer_size 1m;`
Context	stream, server

Sets the maximum size of the response received with Fetch API.

js_fetch_protocols

Syntax	`js_fetch_protocols` [TLSv1] [TLSv1.1] [TLSv1.2] [TLSv1.3];
Default	`js_fetch_protocols TLSv1 TLSv1.1 TLSv1.2;`
Context	stream, server

Enables the specified protocols for HTTPS connections with Fetch API.

js_fetch_timeout

Syntax	`js_fetch_timeout` *time*;
Default	`js_fetch_timeout 60s;`
Context	stream, server

Defines a timeout for reading and writing for Fetch API. The timeout is set only between two successive read/write operations, not for the whole response. If no data is transmitted within this time, the connection is closed.

js_fetch_trusted_certificate

Syntax	`js_fetch_trusted_certificate` *file*;
Default	—
Context	stream, server

Specifies a file with trusted CA certificates in the PEM format used to verify the HTTPS certificate with Fetch API.

js_fetch_verify

Syntax	`js_fetch_verify on | off;`
Default	`js_fetch_verify on;`
Context	stream, server

Enables or disables verification of the HTTPS server certificate with Fetch API.

js_fetch_verify_depth

Syntax	`js_fetch_verify_depth` *number*;
Default	`js_fetch_verify_depth 100;`
Context	stream, server

Sets the verification depth in the HTTPS server certificates chain with Fetch API.

js_filter

Syntax	`js_filter` *function* | *module.function*;
Default	—
Context	stream, server

Sets a data filter. Module functions can be referenced.

The filter function is called once at the moment when the stream session reaches the *content phase*. The filter function is called with the following arguments:

`s`	the stream session object

At this phase, it is possible to perform initialization or register a callback with the s.on() method for each incoming data chunk. The s.off() method may be used to unregister a callback and stop filtering.

Note

As the *js_filter* handler returns its result immediately, it supports only synchronous operations. Thus, asynchronous operations such as ngx.fetch() or setTimeout() are not supported.

js_import

Syntax	`js_import` *module.js* | *export_name from module.js*;
Default	—
Context	stream, server

Imports a module that implements location and variable handlers in njs. The *export_name* is used as a namespace to access module functions. If the *export_name* is not specified, the module name will be used as a namespace.

```
js_import stream.js;
```

Here, the module name *stream* is used as

Several *js_ import* directives can be specified.

js_path

Syntax	`js_path` *path*;
Default	—
Context	stream, server

Sets an additional path for njs modules.

js_preload_object

Syntax	`js_preload_object` *name.json* | *name* from *file.json*;
Default	—
Context	stream, server

Preloads an immutable object at configure time. The *name* is used as a name of the global variable though which the object is available in njs code. If the *name* is not specified, the file name will be used instead.

```
js_preload_object map.json;
```

Here, the *map* is used as a name while accessing the preloaded object.

Several *js_ preload_ object* directives can be specified.

js_preread

Syntax	`js_preread` *function* | *module.function*;
Default	—
Context	stream, server

Sets an njs function which will be called at the *preread phase*. Module functions can be referenced.

The function is called once at the moment when the stream session reaches the *preread phase* for the first time. The function is called with the following arguments:

`s`	the stream session object

At this phase, it is possible to perform initialization or register a callback with the s.on() method for each incoming data chunk until one of the following methods are called: s.done(), s.decline(), s.allow(). When one of these methods is called, the stream session switches to the *next phase* and all current s.on() callbacks are dropped.

Note

As the *js_ preread* handler returns its result immediately, it supports only synchronous callbacks. Thus, asynchronous callbacks such as ngx.fetch() or setTimeout() are not supported. Nevertheless, asynchronous operations are supported in s.on() callbacks in the *preread phase*.

js_set

Syntax	`js_set` *$variable function* | *module.function*;
Default	—
Context	stream, server

Sets an njs function for the specified variable. Module functions can be referenced.

The function is called when the variable is referenced for the first time for a given request. The exact moment depends on a *phase* at which the variable is referenced. This can be used to perform some logic not related to variable evaluation. For example, if the variable is referenced only in the *log_format* directive, its handler will not be executed until the log phase. This handler can be used to do some cleanup right before the request is freed.

Note

As the *js_set* handler returns its result immediately, it supports only synchronous callbacks. Thus, asynchronous callbacks such as ngx.fetch() or setTimeout() are not supported.

js_shared_dict_zone

Syntax	`js_shared_dict_zone zone=`*name:size* [`timeout=`*time*] [`type=`*string* | *number*] [`evict`];
Default	—
Context	stream

Sets the name and size of the shared memory zone that keeps the key-value dictionary shared between worker processes.

`type`	optional parameter, allows redefining the value type to `number`, by default the shared dictionary uses a `string` as a key and a value
`timeout`	optional parameter, sets the time after which all shared dictionary entries are removed from the zone
`evict`	optional parameter, removes the oldest key-value pair when the zone storage is exhausted

Examples:

```
example.conf:
    # Creates a 1Mb dictionary with string values,
    # removes key-value pairs after 60 seconds of inactivity:
    js_shared_dict_zone zone=foo:1M timeout=60s;

    # Creates a 512Kb dictionary with string values,
    # forcibly removes oldest key-value pairs when the zone is exhausted:
    js_shared_dict_zone zone=bar:512K timeout=30s evict;

    # Creates a 32Kb permanent dictionary with number values:
    js_shared_dict_zone zone=num:32k type=number;
```

```
example.js:
    function get(r) {
        r.return(200, ngx.shared.foo.get(r.args.key));
```

```
    }

    function set(r) {
        r.return(200, ngx.shared.foo.set(r.args.key, r.args.value));
    }

    function delete(r) {
        r.return(200, ngx.shared.bar.delete(r.args.key));
    }

    function increment(r) {
        r.return(200, ngx.shared.num.incr(r.args.key, 2));
    }
```

js_var

Syntax	`js_var` *$variable* [*value*];
Default	—
Context	stream, server

Declares a writable variable. The value can contain text, variables, and their combination.

Session Object Properties

Each stream njs handler receives one argument, a stream-session object.

Limit Conn

The module is used to limit the number of connections per the defined key, in particular, the number of connections from a single IP address.

Configuration Example

```
stream {
    limit_conn_zone $binary_remote_addr zone=addr:10m;

    ...

    server {

        ...

        limit_conn           addr 1;
        limit_conn_log_level error;
    }
}
```

Directives

limit_conn

Syntax	`limit_conn` *zone number*;
Default	—
Context	stream, server

Sets the shared memory zone and the maximum allowed number of connections for a given key value. When this limit is exceeded, the server will close the connection. For example, the directives

```
limit_conn_zone $binary_remote_addr zone=addr:10m;

server {
    ...
    limit_conn addr 1;
}
```

allow only one connection per IP address at a time.

When several `limit_conn` directives are specified, any configured limit will apply.

These directives are inherited from the previous configuration level if and only if there are no `limit_conn` directives defined on the current level.

limit_conn_dry_run

Syntax	`limit_conn_dry_run on	off;`
Default	`limit_conn_dry_run off;`
Context	stream, server

Enables the dry run mode. In this mode, the number of connections is not limited, however, in the *shared memory zone*, the number of excessive connections is accounted as usual.

limit_conn_log_level

Syntax	`limit_conn_log_level info	notice	warn	error;`
Default	`limit_conn_log_level error;`
Context	stream, server

Sets the desired logging level for cases when the server limits the number of connections.

limit_conn_zone

Syntax	`limit_conn_zone` *key* zone = *name:size*;
Default	—
Context	stream

Sets parameters for a shared memory zone that will keep states for various keys. In particular, the state includes the current number of connections. The key can contain text, variables, and their combinations. Connections with an empty key value are not accounted.

Usage example:

```
limit_conn_zone $binary_remote_addr zone=addr:10m;
```

Here, a client IP address is set by the `$binary_remote_addr` variable.

The size of `$binary_remote_addr` is 4 bytes for IPv4 addresses or 16 bytes for IPv6 addresses. The stored state always occupies 32 or 64 bytes on 32-bit platforms and 64 bytes on 64-bit platforms.

One megabyte zone can keep about 32 thousand 32-byte states or about 16 thousand 64-byte states. If the zone storage is exhausted, the server will close the connection.

Built-in Variables

`$limit_conn_status`

keeps the result of limiting the number of connections: `PASSED`, `REJECTED` or `REJECTED_DRY_RUN`

Log

The module writes request logs in the specified format.

Configuration Example

```
log_format basic '$remote_addr [$time_local] '
                 '$protocol $status $bytes_sent $bytes_received '
                 '$session_time';

access_log /spool/logs/angie-access.log basic buffer=32k;
```

Directives

access_log

Syntax	`access_log` *path* [*format* [`buffer=`*size*] [gzip[=*level*]] [`flush=`*time*] [`if=`*condition*]]; `access_log off;`
Default	`access_log off;`
Context	stream, server

Sets the `path`, `format`, and configuration for a buffered log write. Several logs can be specified on the same configuration level. Logging to *syslog* can be configured by specifying the `"syslog:"` prefix in the first parameter. The special value `off` cancels all `access_log` directives on the current level.

If either the `buffer` or `gzip` parameter is used, writes to log will be buffered.

Caution

The buffer size must not exceed the size of an atomic write to a disk file. For FreeBSD this size is unlimited.

When buffering is enabled, the data will be written to the file:

- if the next log line does not fit into the buffer;
- if the buffered data is older than specified by the flush parameter;
- when a worker process is *re-opening log files* or is shutting down.

If the *gzip* parameter is used, then the buffered data will be compressed before writing to the file. The compression level can be set between *1* (fastest, less compression) and *9* (slowest, best compression). By default, the buffer size is equal to *64K* bytes, and the compression level is set to *1*. Since the data is compressed in atomic blocks, the log file can be decompressed or read by *"zcat"* at any time.

Example:

```
access_log /path/to/log.gz basic gzip flush=5m;
```

Important

For gzip compression to work, Angie must be built with the zlib library.

The file path can contain variables, but such logs have some constraints:

- the *user* whose credentials are used by worker processes should have permissions to create files in a directory with such logs;
- buffered writes do not work;
- the file is opened and closed for each log write. However, since the descriptors of frequently used files can be stored in a cache, writing to the old file can continue during the time specified by the *open_log_file_cache* directive's *valid* parameter.

The *if* parameter enables conditional logging. A session will not be logged if the condition evaluates to *"0"* or an empty string.

log_format

Syntax	`log_format` *name* [`escape`=default | `json` | `none`] *string* ...;
Default	—
Context	stream, server

Specifies log format.

The `escape` parameter allows setting `json` or `default` characters escaping in variables, by default, `default` escaping is used. The `none` value disables escaping.

For `default` escaping, characters """, "\", and other characters with values less than 32 or above 126 are escaped as "\xXX". If the variable value is not found, a hyphen "-" will be logged.

For `json` escaping, all characters not allowed in JSON strings will be escaped: characters """ and "\" are escaped as "\"" and "\\", characters with values less than 32 are escaped as "\n", "\r", "\t", "\b", "\f", or "\u00XX".

open_log_file_cache

Syntax	`open_log_file_cache` max = *N* [`inactive`=*time*] [`min_uses`=*N*] [`valid`=*time*];
Default	`open_log_file_cache off;`
Context	stream, server

Defines a cache that stores the file descriptors of frequently used logs whose names contain variables. The directive has the following parameters:

`max`	sets the maximum number of descriptors in a cache; if the cache becomes full the least recently used (LRU) descriptors are closed
`inactive`	sets the time after which the cached descriptor is closed if there were no access during this time; by default, 10 seconds
`min_uses`	sets the minimum number of file uses during the time defined by the `inactive` parameter to let the descriptor stay open in a cache; by default, 1
`valid`	sets the time after which it should be checked that the file still exists with the same name; by default, 60 seconds
`off`	disables caching

Usage example:

```
open_log_file_cache max=1000 inactive=20s valid=1m min_uses=2;
```

Map

The module creates variables whose values depend on values of other variables.

Configuration Example

```
map $remote_addr $limit {
    127.0.0.1    "";
    default      $binary_remote_addr;
}

limit_conn_zone $limit zone=addr:10m;
limit_conn addr 1;
```

Directives

map

Syntax	map *string* *$variable* { ... };
Default	—
Context	stream

Creates a new variable. Its value depends on the first parameter, specified as a string with variables, for example:

```
set $var1 "foo";
set $var2 "bar";

map $var1$var2 $new_variable {
    default "foobar_value";
}
```

Here, the variable `$new_variable` will have a value composed of the two variables `$var1` and `$var2`, or a default value if these variables are not defined.

Note

Since variables are evaluated only when they are used, the mere declaration even of a large number of "map" variables does not add any extra costs to request processing.

Parameters inside the *map* block specify a mapping between source and resulting values.

Source values are specified as strings or regular expressions.

Strings are matched ignoring the case.

A regular expression should either start with a "~" symbol for a case-sensitive matching, or with the "~*" symbols for case-insensitive matching. A regular expression can contain named and positional captures that can later be used in other directives along with the resulting variable.

If a source value matches one of the names of special parameters described below, it should be prefixed with the "" symbol.

The resulting value can contain text, variable and their combination.

The following special parameters are also supported:

`default` *value*	sets the resulting value if the source value matches none of the specified variants. When *default* is not specified, the default resulting value will be an empty string.
`hostnames`	indicates that source values can be hostnames with a prefix or suffix mask. This parameter should be specified before the list of values.

For example,

```
*.example.com 1;
example.*     1;
```

The following two records

```
example.com   1;
*.example.com 1;
```

can be combined:

```
.example.com  1;
```

`include` *file*	includes a file with values. There can be several inclusions.
`volatile`	indicates that the variable is not cacheable.

If the source value matches more than one of the specified variants, e.g. both a mask and a regular expression match, the first matching variant will be chosen, in the following order of priority:

1. String value without a mask
2. Longest string value with a prefix mask, e.g. `*.example.com`
3. Longest string value with a suffix mask, e.g. `mail.*`
4. First matching regular expression (in order of appearance in a configuration file)
5. Default value (`default`)

map_hash_bucket_size

Syntax	`map_hash_bucket_size` *size*;
Default	`map_hash_bucket_size 32|64|128;`
Context	stream

Sets the bucket size for the *map* variables hash tables. Default value depends on the processor's cache line size. The details of setting up hash tables are provided *separately*.

map_hash_max_size

Syntax	`map_hash_max_size` *size*;
Default	`map_hash_max_size 2048;`
Context	stream

Sets the maximum size of the *map* variables hash tables. The details of setting up hash tables are provided *separately*.

MQTT Preread

Enables extracting client IDs and usernames from `CONNECT` packets for Message Queuing Telemetry Transport (MQTT) versions 3.1.1 and 5.0.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-stream_mqtt_preread_module` build option.

In packages and images from our repos, the module is included in the build.

Configuration Example

Choosing an upstream server by the client ID:

```
stream {

    mqtt_preread on;

    upstream mqtt {
        hash $mqtt_preread_clientid;
        # ...
    }
}
```

Directives

mqtt_preread

Syntax	`mqtt_preread on | off;`
Default	`mqtt_preread off;`
Context	stream, server

Controls extracting information from `CONNECT` packets during the *preread phase*. If the setting is `on`, its surrounding context will have the following variables populated.

Built-in Variables

See the details of the value semantics in the specification of MQTT versions 3.1.1 and 5.0.

`$mqtt_preread_clientid`

Unique client ID.

`$mqtt_preread_username`

Optional username.

Pass

Allows passing the accepted connection directly to any configured listening socket in *HTTP*, *Stream*, or *Mail* modules.

Configuration Example

After the `stream` module handles the SSL/TLS termination, it forwards the connection to the `http` module:

```
http {
    server {
        listen 8000;

        location / {
            root html;
        }
    }
}

stream {
    server {
        listen 12345 ssl;

        ssl_certificate     domain.crt;
        ssl_certificate_key domain.key;

        pass 127.0.0.1:8000;
    }
}
```

Directives

pass

Syntax	pass *address*;
Default	—
Context	server

This directive sets the server address to which the client connection should be passed. The *address* can be given as an IP address and port:

```
pass 127.0.0.1:12345;
```

Or as a path to a UNIX domain socket:

```
pass unix:/tmp/stream.socket;
```

Also, the *address* can be set with variables:

```
pass $upstream;
```

Proxy

Allows proxying data streams over TCP, UDP, and UNIX domain sockets.

Configuration Example

```
server {
    listen 127.0.0.1:12345;
    proxy_pass 127.0.0.1:8080;
}

server {
    listen 12345;
```

```
    proxy_connect_timeout 1s;
    proxy_timeout 1m;
    proxy_pass example.com:12345;
}

server {
    listen 53 udp reuseport;
    proxy_timeout 20s;
    proxy_pass dns.example.com:53;
}

server {
    listen [::1]:12345;
    proxy_pass unix:/tmp/stream.socket;
}
```

Directives

proxy_bind

Syntax	`proxy_bind` *address* [`transparent`] | `off`;
Default	—
Context	stream, server

Makes outgoing connections to a proxied server originate from the specified local IP address. Parameter value can contain variables. The special value `off` cancels the effect of the *proxy_bind* directive inherited from the previous configuration level, which allows the system to auto-assign the local IP address.

The `transparent` parameter allows outgoing connections to a proxied server originate from a non-local IP address, for example, from a real IP address of a client:

```
proxy_bind $remote_addr transparent;
```

For this parameter to work, Angie worker processes usually need to run with *superuser* privileges. On Linux, this is not required: if the `transparent` parameter is specified, worker processes inherit the *CAP_NET_RAW* capability from the master process.

Important

The kernel routing table should also be configured to intercept network traffic from the FastCGI server.

proxy_buffer_size

Syntax	`proxy_buffer_size` *size*;
Default	`proxy_buffer_size 16k;`
Context	stream, server

Sets the size of the buffer used for reading data from the proxied server. Also sets the size of the buffer used for reading data from the client.

proxy_connect_timeout

Syntax	`proxy_connect_timeout` *time*;
Default	`proxy_connect_timeout 60s;`
Context	stream, server

Defines a timeout for establishing a connection with a proxied server.

proxy_connection_drop

Syntax	`proxy_connection_drop` *time* | `on` | `off`;
Default	`proxy_connection_drop off;`
Context	stream, server

Enables termination of all sessions to the proxied server after it has been removed from the group or marked as permanently unavailable by a *reresolve* process or the *API command* `DELETE`.

A session is terminated when the next read or write event is processed for either the client or the proxied server.

Setting *time* enables a session termination *timeout*; with `on` set, sessions are dropped immediately.

proxy_download_rate

Syntax	`proxy_download_rate` *rate*;
Default	`proxy_download_rate 0;`
Context	stream, server

Limits the speed of reading the data from the proxied server. The `rate` is specified in bytes per second.

`0`	disables rate limiting

Note

The limit is set per a connection, so if Angie simultaneously opens two connections to the proxied server, the overall rate will be twice as much as the specified limit.

Parameter value can contain variables. It may be useful in cases where rate should be limited depending on a certain condition:

```
map $slow $rate {
    1     4k;
    2     8k;
}

proxy_download_rate $rate;
```

proxy_half_close

Syntax	`proxy_half_close on` | `off`;
Default	`proxy_half_close off;`
Context	stream, server

Enables or disables closing each direction of a TCP connection independently ("TCP half-close"). If enabled, proxying over TCP will be kept until both sides close the connection.

proxy_next_upstream

Syntax	`proxy_next_upstream on | off;`
Default	`proxy_next_upstream on;`
Context	stream, server

When a connection to the proxied server cannot be established, determines whether a client connection will be passed to the next server in the *upstream pool*.

Passing a connection to the next server can be limited by the *number of tries* and by *time*.

proxy_next_upstream_timeout

Syntax	`proxy_next_upstream_timeout` *time*;
Default	`proxy_next_upstream_timeout 0;`
Context	stream, server

Limits the time allowed to pass a connection to the *next* server.

`0`	turns off this limitation

proxy_next_upstream_tries

Syntax	`proxy_next_upstream_tries` *number*;
Default	`proxy_next_upstream_tries 0;`
Context	stream, server

Limits the number of possible tries for passing a connection to the *next* server.

`0`	turns off this limitation

proxy_pass

Syntax	`proxy_pass` *address*;
Default	—
Context	server

Sets the address of a proxied server. The `address` can be specified as a domain name or IP address, and a port:

```
proxy_pass localhost:12345;
```

or as a UNIX domain socket path:

```
proxy_pass unix:/tmp/stream.socket;
```

If a domain name resolves to several addresses, all of them will be used in a round-robin fashion. In addition, an address can be specified as a *server group*.

The address can also be specified using variables:

```
proxy_pass $upstream;
```

In this case, the server name is searched among the described *server groups* and, if not found, is determined using a *resolver*.

proxy_protocol

Syntax	`proxy_protocol on	off;`
Default	`proxy_protocol off;`
Context	stream, server

Enables the PROXY protocol for connections to a proxied server.

proxy_requests

Syntax	`proxy_requests` *number*;
Default	`proxy_requests 0;`
Context	stream, server

Sets the number of client datagrams at which binding between a client and existing UDP stream session is dropped. After receiving the specified number of datagrams, next datagram from the same client starts a new session. The session terminates when all client datagrams are transmitted to a proxied server and the expected *number of responses* is received, or when it reaches a *timeout*.

proxy_responses

Syntax	`proxy_responses` *number*;
Default	—
Context	stream, server

Sets the number of datagrams expected from the proxied server in response to a client datagram if the *UDP* protocol is used. The number serves as a hint for session termination. By default, the number of datagrams is not limited.

If zero value is specified, no response is expected. However, if a response is received and the session is still not finished, the response will be handled.

proxy_socket_keepalive

Syntax	`proxy_socket_keepalive on	off;`
Default	`proxy_socket_keepalive off;`
Context	stream, server

Configures the "TCP keepalive" behavior for outgoing connections to a proxied server.

`""`	By default, the operating system's settings are in effect for the socket.
`on`	The SO_KEEPALIVE socket option is turned on for the socket.

proxy_ssl

Syntax	`proxy_ssl on	off;`
Default	`proxy_ssl off;`
Context	stream, server

Enables the SSL/TLS protocol for connections to a proxied server.

proxy_ssl_certificate

Syntax	`proxy_ssl_certificate` *file* [*file*];
Default	—
Context	stream, server

Specifies a file with the certificate in the PEM format used for authentication to a proxied server. Variables can be used in the file name.

Added in version 1.2.0.

When *proxy_ssl_ntls* enabled, directive accepts two arguments instead of one:

```
server {
    proxy_ssl_ntls  on;

    proxy_ssl_certificate      sign.crt enc.crt;
    proxy_ssl_certificate_key  sign.key enc.key;

    proxy_ssl_ciphers "ECC-SM2-WITH-SM4-SM3:ECDHE-SM2-WITH-SM4-SM3:RSA";

    proxy_pass backend:12345;
}
```

proxy_ssl_certificate_key

Syntax	`proxy_ssl_certificate_key` *file* [*file*];
Default	—
Context	stream, server

Specifies a file with the secret key in the PEM format used for authentication to a proxied server. Variables can be used in the file name.

Added in version 1.2.0.

When *proxy_ssl_ntls* enabled, directive accepts two arguments instead of one:

```
server {
    proxy_ssl_ntls  on;

    proxy_ssl_certificate      sign.crt enc.crt;
    proxy_ssl_certificate_key  sign.key enc.key;

    proxy_ssl_ciphers "ECC-SM2-WITH-SM4-SM3:ECDHE-SM2-WITH-SM4-SM3:RSA";

    proxy_pass backend:12345;
}
```

proxy_ssl_ciphers

Syntax	`proxy_ssl_ciphers` *ciphers*;
Default	`proxy_ssl_ciphers DEFAULT;`
Context	stream, server

Specifies the enabled ciphers for requests to a proxied server. The ciphers are specified in the format understood by the OpenSSL library.

The list of ciphers depends on the version of OpenSSL installed. The full list can be viewed using the `openssl ciphers` command.

Attention

The `proxy_ssl_ciphers` directive does *not* configure ciphers for TLS 1.3 when using OpenSSL. To tune TLS 1.3 ciphers with OpenSSL, use the *proxy_ssl_conf_command* directive, which was added to support advanced SSL configuration.

- In LibreSSL, TLS 1.3 ciphers *can* be configured using `proxy_ssl_ciphers`.
- In BoringSSL, TLS 1.3 ciphers cannot be configured at all.

proxy_ssl_conf_command

Syntax	`proxy_ssl_conf_command` *name value*;
Default	—
Context	stream, server

Sets arbitrary OpenSSL configuration commands when establishing a connection with the proxied server.

Important

The directive is supported when using OpenSSL 1.0.2 or higher. To configure TLS 1.3 ciphers with OpenSSL, use the `ciphersuites` command.

Several *proxy_ssl_conf_command* directives can be specified on the same level. These directives are inherited from the previous configuration level if and only if there are no *proxy_ssl_conf_command* directives defined on the current level.

Caution

Note that configuring OpenSSL directly might result in unexpected behavior.

proxy_ssl_crl

Syntax	`proxy_ssl_crl` *file*;
Default	—
Context	stream, server

Specifies a file with revoked certificates (CRL) in the PEM format used to *verify* the certificate of the proxied server.

proxy_ssl_name

Syntax	`proxy_ssl_name` *name*;
Default	`proxy_ssl_name host from proxy_pass;`
Context	stream, server

Allows overriding the server name used to *verify* the certificate of the proxied server and to be *passed through SNI* when establishing a connection with the proxied server. The server name can also be specified using variables.

By default, the host part of the address specified in the *proxy_pass* directive is used.

proxy_ssl_ntls

Added in version 1.2.0.

Syntax	`proxy_ssl_ntls on | off;`
Default	`proxy_ssl_ntls off;`
Context	stream, server

Enables client-side support for NTLS using the TongSuo TLS library.

```
server {
    proxy_ssl_ntls  on;

    proxy_ssl_certificate      sign.crt enc.crt;
    proxy_ssl_certificate_key  sign.key enc.key;

    proxy_ssl_ciphers "ECC-SM2-WITH-SM4-SM3:ECDHE-SM2-WITH-SM4-SM3:RSA";

    proxy_pass backend:12345;
}
```

Important

Build Angie using the *--with-ntls* build option and link with NTLS-enabled SSL library

```
./configure --with-openssl=../Tongsuo-8.3.0 \
            --with-openssl-opt=enable-ntls  \
            --with-ntls
```

proxy_ssl_password_file

Syntax	`proxy_ssl_password_file` *file*;
Default	—
Context	stream, server

Specifies a file with passphrases for *secret keys* where each passphrase is specified on a separate line. Passphrases are tried in turn when loading the key.

proxy_ssl_protocols

Syntax	`proxy_ssl_protocols [SSLv2] [SSLv3] [TLSv1] [TLSv1.1] [TLSv1.2] [TLSv1.3];`
Default	`proxy_ssl_protocols TLSv1.2 TLSv1.3;`
Context	stream, server

Changed in version 1.2.0: `TLSv1.3` parameter added to default set.

Enables the specified protocols for requests to a proxied server.

proxy_ssl_server_name

Syntax	`proxy_ssl_server_name on | off;`
Default	`proxy_ssl_server_name off;`
Context	stream, server

Enables or disables passing the server name set by the *proxy_ssl_name* directive via the Server Name Indication TLS extension (SNI, RFC 6066) while establishing a connection with the proxied server.

proxy_ssl_session_reuse

Syntax	`proxy_ssl_session_reuse on | off;`
Default	`proxy_ssl_session_reuse on;`
Context	stream, server

Determines whether SSL sessions can be reused when working with the proxied server. If the errors "*SSL3_GET_FINISHED:digest check failed*" appear in the logs, try disabling session reuse.

proxy_ssl_trusted_certificate

Syntax	`proxy_ssl_trusted_certificate` *file*;
Default	—
Context	stream, server

Specifies a file with trusted CA certificates in the PEM format used to *verify* the certificate of the proxied server.

proxy_ssl_verify

Syntax	`proxy_ssl_verify on | off;`
Default	`proxy_ssl_verify off;`
Context	stream, server

Enables or disables verification of the proxied server certificate.

proxy_ssl_verify_depth

Syntax	`proxy_ssl_verify_depth` *number*;
Default	`proxy_ssl_verify_depth 1;`
Context	stream, server

Sets the verification depth in the proxied server certificates chain.

proxy_timeout

Syntax	`proxy_timeout` *time*;
Default	`proxy_timeout 10m;`
Context	stream, server

Sets the timeout between two successive read or write operations on client or proxied server connections. If no data is transmitted within this time, the connection is closed.

upstream_probe_timeout (PRO)

Added in version 1.4.0: PRO

Syntax	`upstream_probe_timeout` *time*;
Default	`upstream_probe_timeout 50s;`
Context	server

Sets the maximum inactivity *time* of an established server connection for probes configured using the *upstream_probe (PRO)* directive; if this limit is exceeded, the connection will be closed.

proxy_upload_rate

Syntax	`proxy_upload_rate` *rate*;
Default	`proxy_upload_rate 0;`
Context	stream, server

Limits the speed of reading the data from the client. The `rate` is specified in bytes per second.

`0`	disables rate limiting

Note

The limit is set per a connection, so if the client simultaneously opens two connections, the overall rate will be twice as much as the specified limit.

Parameter value can contain variables. It may be useful in cases where rate should be limited depending on a certain condition:

```
map $slow $rate {
    1     4k;
    2     8k;
}

proxy_upload_rate $rate;
```

RDP Preread

When using the RDP protocol, this module allows extracting cookies, which are used for session identification and management, before making a load balancing decision.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-stream_rdp_preread_module` build option.

In packages and images from our repos, the module is included in the build.

Configuration Example

Binding to the Cookie-Issuing Server

This uses the `learn` mode of the *sticky* directive:

```
stream {

    rdp_preread on;

    upstream rdp {

        server 127.0.0.1:3390 sid=a;
        server 127.0.0.1:3391 sid=b;

        sticky learn lookup=$rdp_cookie create=$rdp_cookie zone=sessions:1m;
    }
}
```

Directives

rdp_preread

Syntax	`rdp_preread on | off;`
Default	`rdp_preread off;`
Context	stream, server

Controls extracting information from RDP protocol cookies during the *preread stage*. If the setting is `on`, its surrounding context will have the following variables populated.

Built-in Variables

The semantics of cookie values depend on the RDP protocol version.

`$rdp_cookie`

The entire cookie value.

`$rdp_cookie_<name>`

The value of the cookie field with the specified `name`.

RealIP

The module is used to change the client address and port to the ones sent in the PROXY protocol header. The PROXY protocol must be previously enabled by setting the *proxy_protocol* parameter in the *listen* directive.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-stream_realip_module` build option.

In packages and images from our repos, the module is included in the build.

Configuration Example

```
listen 12345 proxy_protocol;

set_real_ip_from  192.168.1.0/24;
set_real_ip_from  192.168.2.1;
set_real_ip_from  2001:0db8::/32;
```

Directives

set_real_ip_from

Syntax	`set_real_ip_from` *address* | *CIDR* | `unix:`;
Default	—
Context	stream, server

Defines trusted addresses that are known to send correct replacement addresses. If the special value `unix:` is specified, all UNIX domain sockets will be trusted.

Built-in Variables

`$realip_remote_addr`

keeps the original client address

`$realip_remote_port`

keeps the original client port

Return

The module allows sending a specified value to the client and then closing the connection.

Configuration Example

```
server {
    listen 12345;
    return $time_iso8601;
}
```

Directives

return

Syntax	`return` *value*;
Default	—
Context	server

Specifies a value to send to the client. The value can contain text, variables, and their combination.

Set

The module allows setting a value for a variable.

Configuration Example

```
server {
    listen 12345;
    set    $true 1;
}
```

Directives

set

Syntax	`set` *$variable value*;
Default	—
Context	server

Sets a value for the specified variable. The value can contain text, variables, and their combination.

Split Clients

The module emits variables for A/B testing, canary releases, or other scenarios that require routing a percentage of clients to one server or configuration while directing the rest elsewhere.

Configuration Example

```
stream {
    # ...
    split_clients "${remote_addr}AAA" $upstream {
                  0.5%                feature_test1;
                  2.0%                feature_test2;
                  *                   production;
    }

    server {
        # ...
        proxy_pass $upstream;
    }
}
```

Directives

split_clients

Syntax	`split_clients` *string $variable* { ... }
Default	—
Context	stream

Creates a *$variable* by hashing the *string*; the variables in *string* are substituted, the result is hashed, then the hash is mapped to the *$variable*'s string value.

The hash function uses MurmurHash2 (32-bit), and its entire value range (0 to 4294967295) is mapped to buckets in order of appearance; the percentages determine the size of the buckets. A wildcard (`*`) may occur last; hashes that fall outside other buckets are mapped to its assigned value.

An example:

```
split_clients "${remote_addr}AAA" $variant {
               0.5%               .one;
               2.0%               .two;
               *                  "";
}
```

Here, the hashed values of the interpolated `$remote_addrAAA` string are distributed as follows:

- values 0 to 21474835 (a sample of 0.5%) yield `.one`
- values 21474836 to 107374180 (a sample of 2%) yield `.two`
- values 107374181 to 4294967295 (all other values) yield `""` (an empty string)

SSL

Provides the necessary support for a stream proxy server to work with the SSL/TLS protocol.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-stream_ssl_module` build option.

In packages and images from our repos, the module is included in the build.

Important

This module requires the OpenSSL library.

Configuration Example

To reduce the processor load it is recommended to

- set the number of *worker processes* equal to the number of processors,
- enable the *shared* session cache,
- disable the *built-in* session cache,
- and possibly increase the session *lifetime* (by default, 5 minutes):

```
worker_processes auto;

stream {

    #...

    server {
        listen              12345 ssl;

        ssl_protocols       TLSv1.2 TLSv1.3;
        ssl_ciphers         AES128-SHA:AES256-SHA:RC4-SHA:DES-CBC3-SHA:RC4-MD5;
        ssl_certificate     /usr/local/angie/conf/cert.pem;
        ssl_certificate_key /usr/local/angie/conf/cert.key;
        ssl_session_cache   shared:SSL:10m;
        ssl_session_timeout 10m;
```

```
    #    ...
    }
```

Directives

ssl_alpn

Syntax	`ssl_alpn` *protocol* ...;
Default	—
Context	stream, server

Specifies the list of supported ALPN protocols. One of the protocols must be *negotiated* if the client uses ALPN:

```
map $ssl_alpn_protocol $proxy {
    h2                 127.0.0.1:8001;
    http/1.1           127.0.0.1:8002;
}

server {
    listen      12346;
    proxy_pass  $proxy;
    ssl_alpn    h2 http/1.1;
}
```

ssl_certificate

Syntax	`ssl_certificate` *file*;
Default	—
Context	stream, server

Specifies a file with the certificate in the PEM format for the given server. If intermediate certificates should be specified in addition to a primary certificate, they should be specified in the same file in the following order: the primary certificate comes first, then the intermediate certificates. A secret key in the PEM format may be placed in the same file.

This directive can be specified multiple times to load certificates of different types, for example, RSA and ECDSA:

```
server {
    listen              12345 ssl;

    ssl_certificate     example.com.rsa.crt;
    ssl_certificate_key example.com.rsa.key;

    ssl_certificate     example.com.ecdsa.crt;
    ssl_certificate_key example.com.ecdsa.key;

#    ...
}
```

Only OpenSSL 1.0.2 or higher supports separate certificate chains for different certificates. With older versions, only one certificate chain can be used.

Important

Variables can be used in the file name when using OpenSSL 1.0.2 or higher:

```
ssl_certificate     $ssl_server_name.crt;
ssl_certificate_key $ssl_server_name.key;
```

Note that using variables implies that a certificate will be loaded for each SSL handshake, and this may have a negative impact on performance.

The value "data:*$variable*" can be specified instead of the `file`, which loads a certificate from a variable without using intermediate files.

Note that inappropriate use of this syntax may have its security implications, such as writing secret key data to *error log*.

ssl_certificate_key

Syntax	`ssl_certificate_key` *file*;
Default	—
Context	stream, server

Specifies a file with the secret key in the PEM format for the given server.

Important

Variables can be used in the file name when using OpenSSL 1.0.2 or higher.

The value "engine:*name*:id" can be specified instead of the `file`, which loads a secret key with a specified *id* from the OpenSSL engine *name*.

The value "data:*$variable*" can be specified instead of the `file`, which loads a secret key from a variable without using intermediate files. Note that inappropriate use of this syntax may have its security implications, such as writing secret key data to *error log*.

ssl_ciphers

Syntax	`ssl_ciphers` *ciphers*;
Default	`ssl_ciphers HIGH:!aNULL:!MD5;`
Context	stream, server

Specifies the enabled ciphers. The ciphers are specified in the format understood by the OpenSSL library, for example:

```
ssl_ciphers ALL:!aNULL:!EXPORT56:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP;
```

The list of ciphers depends on the version of OpenSSL installed. The full list can be viewed using the `openssl ciphers` command.

Attention

The `ssl_ciphers` directive does *not* configure ciphers for TLS 1.3 when using OpenSSL. To tune TLS 1.3 ciphers with OpenSSL, use the *ssl_conf_command* directive, which was added to support advanced SSL configuration.

- In LibreSSL, TLS 1.3 ciphers *can* be configured using `ssl_ciphers`.
- In BoringSSL, TLS 1.3 ciphers cannot be configured at all.

ssl_client_certificate

Syntax	`ssl_client_certificate` *file*;
Default	—
Context	stream, server

Specifies a file with trusted CA certificates in the PEM format used to *verify* client certificates and OCSP responses if *ssl_stapling* is enabled.

The list of certificates will be sent to clients. If this is not desired, the *ssl_trusted_certificate* directive can be used.

ssl_conf_command

Syntax	`ssl_conf_command` *name value*;
Default	—
Context	stream, server

Sets arbitrary OpenSSL configuration commands.

Important

The directive is supported when using OpenSSL 1.0.2 or higher. To configure TLS 1.3 ciphers with OpenSSL, use the `ciphersuites` command.

Several *ssl_conf_command* directives can be specified on the same level:

```
ssl_conf_command Options PrioritizeChaCha;
ssl_conf_command Ciphersuites TLS_CHACHA20_POLY1305_SHA256;
```

These directives are inherited from the previous configuration level if and only if there are no *ssl_conf_command* directives defined on the current level.

Caution

Note that configuring OpenSSL directly might result in unexpected behavior.

ssl_crl

Syntax	`ssl_crl` *file*;
Default	—
Context	stream, server

Specifies a file with revoked certificates (CRL) in the PEM format used to *verify* client certificates.

ssl_dhparam

Syntax	`ssl_dhparam` *file*;
Default	—
Context	stream, server

Specifies a file with DH parameters for DHE ciphers.

Caution

By default no parameters are set, and therefore DHE ciphers will not be used.

ssl_early_data

Syntax	`ssl_early_data on | off;`
Default	`ssl_early_data off;`
Context	stream, server

Enables or disables TLS 1.3 early data.

Important

The directive is supported when using OpenSSL 1.1.1 or higher or BoringSSL.

ssl_ecdh_curve

Syntax	`ssl_ecdh_curve` *curve*;
Default	`ssl_ecdh_curve auto;`
Context	stream, server

Specifies a curve for ECDHE ciphers.

Important

When using OpenSSL 1.0.2 or higher, it is possible to specify multiple curves, for example:

```
ssl_ecdh_curve prime256v1:secp384r1;
```

The special value `auto` instructs Angie to use a list built into the OpenSSL library when using OpenSSL 1.0.2 or higher, or prime256v1 with older versions.

Important

When using OpenSSL 1.0.2 or higher, this directive sets the list of curves supported by the server. Thus, in order for ECDSA certificates to work, it is important to include the curves used in the certificates.

ssl_handshake_timeout

Syntax	`ssl_handshake_timeout` *time*;
Default	`ssl_handshake_timeout 60s;`
Context	stream, server

Specifies a timeout for the SSL handshake to complete.

ssl_ocsp

Syntax	`ssl_ocsp on` | `off` | `leaf`;
Default	`ssl_ocsp off;`
Context	http, server

Enables OCSP validation of the client certificate chain. The `leaf` parameter enables validation of the client certificate only.

For the OCSP validation to work, the *ssl_verify_client* directive should be set to `on` or `optional`.

To resolve the OCSP responder hostname, the *resolver* directive should also be specified.

Example:

```
ssl_verify_client on;
ssl_ocsp          on;
resolver          127.0.0.53;
```

ssl_ocsp_cache

Syntax	`ssl_ocsp_cache off` | [shared:*name*:*size*];
Default	`ssl_ocsp_cache off;`
Context	http, server

Sets name and size of the cache that stores client certificates status for OCSP validation. The cache is shared between all worker processes. A cache with the same name can be used in several virtual servers.

The `off` parameter prohibits the use of the cache.

ssl_ocsp_responder

Syntax	`ssl_ocsp_responder` *uri*;
Default	—
Context	http, server

Overrides the URI of the OCSP responder specified in the "Authority Information Access" certificate extension for *validation* of client certificates.

Only `http://` OCSP responders are supported:

```
ssl_ocsp_responder http://ocsp.example.com/;
```

ssl_ntls

Added in version 1.2.0.

Syntax	`ssl_ntls on	off;`
Default	`ssl_ntls off;`
Context	stream, server

Enables server-side support for NTLS using TongSuo library.

```
listen ... ssl;
ssl_ntls  on;
```

Important

Build Angie using the *--with-ntls* build option and link with NTLS-enabled SSL library

```
./configure --with-openssl=../Tongsuo-8.3.0 \
            --with-openssl-opt=enable-ntls  \
            --with-ntls
```

ssl_password_file

Syntax	`ssl_password_file` *file*;
Default	—
Context	stream, server

Specifies a file with passphrases for *secret keys* where each passphrase is specified on a separate line. Passphrases are tried in turn when loading the key.

Example:

```
stream {
    ssl_password_file /etc/keys/global.pass;
    ...

    server {
        listen 127.0.0.1:12345;
        ssl_certificate_key /etc/keys/first.key;
    }

    server {
        listen 127.0.0.1:12346;

        # named pipe can also be used instead of a file
        ssl_password_file /etc/keys/fifo;
        ssl_certificate_key /etc/keys/second.key;
    }
}
```

ssl_prefer_server_ciphers

Syntax	`ssl_prefer_server_ciphers on	off;`
Default	`ssl_prefer_server_ciphers off;`
Context	stream, server

Specifies that server ciphers should be preferred over client ciphers when the SSLv3 and TLS protocols are used.

ssl_protocols

Syntax	`ssl_protocols` [SSLv2] [SSLv3] [TLSv1] [TLSv1.1] [TLSv1.2] [TLSv1.3];
Default	`ssl_protocols TLSv1.2 TLSv1.3;`
Context	stream, server

Changed in version 1.2.0: `TLSv1.3` parameter added to default set.

Enables the specified protocols.

Important

The TLSv1.1 and TLSv1.2 parameters work only when OpenSSL 1.0.1 or higher is used.

The TLSv1.3 parameter works only when OpenSSL 1.1.1 or higher is used.

ssl_session_cache

Syntax	`ssl_session_cache off	none |` [builtin[:*size*]] [shared:*name*:*size*];
Default	`ssl_session_cache none;`
Context	stream, server

Sets the types and sizes of caches that store session parameters. A cache can be of any of the following types:

`off`	the use of a session cache is strictly prohibited: Angie explicitly tells a client that sessions may not be reused.
`none`	the use of a session cache is gently disallowed: Angie tells a client that sessions may be reused, but does not actually store session parameters in the cache.
`builtin`	a cache built in OpenSSL; used by one worker process only. The cache size is specified in sessions. If size is not given, it is equal to 20480 sessions. Use of the built-in cache can cause memory fragmentation.
`shared`	a cache shared between all worker processes. The cache size is specified in bytes; one megabyte can store about 4000 sessions. Each shared cache should have an arbitrary name. A cache with the same name can be used in several servers. It is also used to automatically generate, store, and periodically rotate TLS session ticket keys unless configured explicitly using the *ssl_session_ticket_key* directive.

Both cache types can be used simultaneously, for example:

```
ssl_session_cache builtin:1000 shared:SSL:10m;
```

but using only shared cache without the built-in cache should be more efficient.

ssl_session_ticket_key

Syntax	`ssl_session_ticket_key` *file*;
Default	—
Context	stream, server

Sets a file with the secret key used to encrypt and decrypt TLS session tickets. The directive is necessary if the same key has to be shared between multiple servers. By default, a randomly generated key is used.

If several keys are specified, only the first key is used to encrypt TLS session tickets. This allows configuring key rotation, for example:

```
ssl_session_ticket_key current.key;
ssl_session_ticket_key previous.key;
```

The file must contain 80 or 48 bytes of random data and can be created using the following command:

```
openssl rand 80 > ticket.key
```

Depending on the file size either AES256 (for 80-byte keys) or AES128 (for 48-byte keys) is used for encryption.

ssl_session_tickets

Syntax	`ssl_session_tickets on | off;`
Default	`ssl_session_tickets on;`
Context	stream, server

Enables or disables session resumption through TLS session tickets.

ssl_session_timeout

Syntax	`ssl_session_timeout` *time*;
Default	`ssl_session_timeout 5m;`
Context	stream, server

Specifies a time during which a client may reuse the session parameters.

ssl_stapling

Syntax	`ssl_stapling on | off;`
Default	`ssl_stapling off;`
Context	http, server

Enables or disables stapling of OCSP responses by the server. Example:

```
ssl_stapling on;
resolver 127.0.0.53;
```

For the OCSP stapling to work, the certificate of the server certificate issuer should be known. If the *ssl_certificate* file does not contain intermediate certificates, the certificate of the server certificate issuer should be present in the *ssl_trusted_certificate* file.

> **Attention**
>
> For the resolution of the OCSP responder hostname, the *resolver* directive should also be specified.

ssl_stapling_file

Syntax	`ssl_stapling_file` *file*;
Default	—
Context	http, server

When set, the stapled OCSP response will be taken from the specified file instead of querying the OCSP responder specified in the server certificate.

The file should be in the DER format as produced by the `openssl ocsp` command.

ssl_stapling_responder

Syntax	`ssl_stapling_responder uri`;
Default	—
Context	http, server

Overrides the URI of the OCSP responder specified in the "Authority Information Access" certificate extension.

Only `http://` OCSP responders are supported:

```
ssl_stapling_responder http://ocsp.example.com/;
```

ssl_stapling_verify

Syntax	`ssl_stapling_verify on` | `off`;
Default	`ssl_stapling_verify off;`
Context	http, server

Enables or disables verification of OCSP responses by the server.

For verification to work, the certificate of the server certificate issuer, the root certificate, and all intermediate certificates should be configured as trusted using the *ssl_trusted_certificate* directive.

ssl_trusted_certificate

Syntax	`ssl_trusted_certificate` *file*;
Default	—
Context	stream, server

Specifies a file with trusted CA certificates in the PEM format used to *verify* client certificates.

In contrast to the certificate set by *ssl_client_certificate*, the list of these certificates will not be sent to clients.

ssl_verify_client

Syntax	`ssl_verify_client on | off | optional | optional_no_ca;`
Default	`ssl_verify_client off;`
Context	stream, server

Enables verification of client certificates. The verification result is stored in the *$ssl_client_verify* variable. If an error has occurred during the client certificate verification or a client has not presented the required certificate, the connection is closed.

`optional`	requests the client certificate and verifies it if the certificate is present.
`optional_no_ca`	requests the client certificate but does not require it to be signed by a trusted CA certificate. This is intended for the use in cases when a service that is external to Angie performs the actual certificate verification.

ssl_verify_depth

Syntax	`ssl_verify_depth` *number*;
Default	`ssl_verify_depth 1;`
Context	stream, server

Sets the verification depth in the client certificates chain.

Built-in Variables

The `stream_ssl` module supports the following variables:

`$ssl_alpn_protocol`

returns the protocol selected by ALPN during the SSL handshake, or an empty string otherwise.

`$ssl_cipher`

returns the name of the cipher used for an established SSL connection.

`$ssl_ciphers`

returns the list of ciphers supported by the client. Known ciphers are listed by names, unknown are shown in hexadecimal, for example:

AES128-SHA:AES256-SHA:0x00ff

Important

The variable is fully supported only when using OpenSSL version 1.0.2 or higher. With older versions, the variable is available only for new sessions and lists only known ciphers.

`$ssl_client_cert`

returns the client certificate in the PEM format for an established SSL connection, with each line except the first prepended with the tab character.

`$ssl_client_fingerprint`

returns the SHA1 fingerprint of the client certificate for an established SSL connection.

`$ssl_client_i_dn`

returns the "issuer DN" string of the client certificate for an established SSL connection according to RFC 2253.

`$ssl_client_raw_cert`

returns the client certificate in the PEM format for an established SSL connection.

`$ssl_client_s_dn`

returns the "subject DN" string of the client certificate for an established SSL connection according to RFC 2253.

`$ssl_client_serial`

returns the serial number of the client certificate for an established SSL connection.

`$ssl_client_v_end`

returns the end date of the client certificate.

`$ssl_client_v_remain`

returns the number of days until the client certificate expires.

`$ssl_client_v_start`

returns the start date of the client certificate.

`$ssl_client_verify`

returns the result of client certificate verification: `SUCCESS`, `FAILED:reason` and `NONE` if a certificate was not present.

`$ssl_curve`

returns the negotiated curve used for SSL handshake key exchange process. Known curves are listed by names, unknown are shown in hexadecimal, for example:

prime256v1

Important

The variable is supported only when using OpenSSL version 3.0 or higher. With older versions, the variable value will be an empty string.

`$ssl_curves`

returns the list of curves supported by the client. Known curves are listed by names, unknown are shown in hexadecimal, for example:

0x001d:prime256v1:secp521r1:secp384r1

Important

The variable is supported only when using OpenSSL version 1.0.2 or higher. With older versions, the variable value will be an empty string.

The variable is available only for new sessions.

`$ssl_early_data`

returns "1" if TLS 1.3 *early data* is used and the handshake is not complete, otherwise "".

`$ssl_protocol`

returns the protocol of an established SSL connection.

`$ssl_server_cert_type`

takes the values `RSA`, `DSA`, `ECDSA`, `ED448`, `ED25519`, `SM2`, `RSA-PSS`, or `unknown` depending on the type of server certificate and key.

`$ssl_server_name`

returns the server name requested through SNI.

`$ssl_session_id`

returns the session identifier of an established SSL connection.

`$ssl_session_reused`

returns `r` if an SSL session was reused, or "." otherwise

SSL Preread

Enables extracting information from the ClientHello message without terminating SSL/TLS, such as the server name requested via SNI or protocols advertised in ALPN.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-stream_ssl_preread_module` build option.

In packages and images from our repos, the module is included in the build.

Configuration Example

Selecting an upstream by server name

```
map $ssl_preread_server_name $name {
    backend.example.com      backend;
    default                  backend2;
}

upstream backend {
    server 192.168.0.1:12345;
    server 192.168.0.2:12345;
}

upstream backend2 {
    server 192.168.0.3:12345;
```

```
    server 192.168.0.4:12345;
}

server {
    listen      12346;
    proxy_pass  $name;
    ssl_preread on;
}
```

Selecting a server by protocol

```
map $ssl_preread_alpn_protocols $proxy {
    ~\bh2\b           127.0.0.1:8001;
    ~\bhttp/1.1\b     127.0.0.1:8002;
    ~\bxmpp-client\b  127.0.0.1:8003;
}

server {
    listen      9000;
    proxy_pass  $proxy;
    ssl_preread on;
}
```

Selecting a server by SSL version

```
map $ssl_preread_protocol $upstream {
    ""         ssh.example.com:22;
    "TLSv1.2" new.example.com:443;
    default    tls.example.com:443;
}

# ssh and https at the same port
server {
    listen      192.168.0.1:443;
    proxy_pass  $upstream;
    ssl_preread on;
}
```

Directives

ssl_preread

Syntax	ssl_preread on | off;
Default	ssl_preread off;
Context	stream, server

Enables extracting information from the ClientHello message at the *preread* phase.

Built-in Variables

`$ssl_preread_protocol`

Highest SSL version supported by the client.

`$ssl_preread_server_name`

Server name requested via SNI.

`$ssl_preread_alpn_protocols`

List of protocols advertised by the client through ALPN. The values are comma separated.

Upstream

The module is used to define groups of servers that can be referenced by the *proxy_pass* directive.

Configuration Example

```
upstream backend {
    hash $remote_addr consistent;
    zone backend 1m;

    server backend1.example.com:1935  weight=5;
    server unix:/tmp/backend3;
    server backend3.example.com       service=_example._tcp resolve;

    server backup1.example.com:1935   backup;
    server backup2.example.com:1935   backup;
}

resolver 127.0.0.53 status_zone=resolver;

server {
    listen 1936;
    proxy_pass backend;
}
```

Directives

upstream

Syntax	`upstream` *name* { ... }
Default	—
Context	stream

Defines a group of servers. Servers can listen on different ports. In addition, servers listening on TCP and UNIX domain sockets can be mixed.

Example:

```
upstream backend {
    server backend1.example.com:1935 weight=5;
    server 127.0.0.1:1935            max_fails=3 fail_timeout=30s;
    server unix:/tmp/backend2;
    server backend3.example.com:1935 resolve;

    server backup1.example.com:1935  backup;
}
```

By default, requests are distributed between the servers using a weighted round-robin balancing

method. In the above example, each 7 requests will be distributed as follows: 5 requests go to backend1.example.com and one request to each of the second and third servers.

If an error occurs during communication with a server, the request will be passed to the next server, and so on until all of the functioning servers will be tried. If a successful response could not be obtained from any of the servers, the client will receive the result of the communication with the last server.

server

Syntax	`server` *address* [*parameters*];
Default	—
Context	upstream

Defines the address and other parameters of a server. The address can be specified as a domain name or IP address with an obligatory port, or as a UNIX domain socket path specified after the `unix:` prefix. A domain name that resolves to several IP addresses defines multiple servers at once.

The following parameters can be defined:

`weight=`*number*	sets the weight of the server; by default, 1.
`max_conns=`*number*	limits the maximum number of simultaneous active connections to the proxied server. Default value is `0`, meaning there is no limit. If the server group does not reside in the *shared memory*, the limitation works per each worker process.

`max_fails=`*number* — sets the number of unsuccessful attempts to communicate with the server that should happen in the duration set by *fail_timeout* to consider the server unavailable; it is then retried after the same duration.

Here, an unsuccessful attempt is an error or timeout while establishing a connection with the server.

> **Note**
>
> If a `server` in an upstream resolves into multiple peers, its `max_fails` setting applies to each peer individually.
>
> If an upstream contains only one peer after all its `server` directives are resolved, the `max_fails` setting has no effect and will be ignored.

`max_fails=1`	the default number of unsuccessful attempts
`max_fails=0`	disables the accounting of attempts

`fail_timeout=`*time* — sets the period of time during which a number of unsuccessful attempts to communicate with the server (*max_fails*) should happen to consider the server unavailable. The server then becomes unavailable for the same amount of time before it is retried.

By default, this is set to 10 seconds.

> **Note**
>
> If a `server` in an upstream resolves into multiple peers, its `fail_timeout` setting applies to each peer individually.
>
> If an upstream contains only one peer after all its `server` directives are resolved, the `fail_timeout` setting has no effect and will be ignored.

`backup`	marks the server as a backup server. It will be passed requests when the primary servers are unavailable.
`down`	marks the server as permanently unavailable.
`drain` (PRO)	sets the server to draining; this means it receives only requests from the sessions that were bound earlier with *sticky*. Otherwise it behaves similarly to `down`.

Caution

The `backup` parameter cannot be used along with the *hash* and *random* load balancing methods.

The `down` and `drain` options are mutually exclusive.

Added in version 1.3.0.

`resolve`	Enables monitoring changes to the list of IP addresses that corresponds to a domain name, updating it without a configuration reload. The group should be stored in a *shared memory zone*; also, you need to define a *resolver*.
`service=`*name*	Enables resolving DNS SRV records and sets the service name. For this parameter to work, specify the *resolve* server parameter, providing a hostname without a port number. If there are no dots in the service name, the name is formed according to the RFC standard: the service name is prefixed with `_`, then `_tcp` is added after a dot. Thus, the service name `http` will result in `_http._tcp`. Angie resolves the SRV records by combining the normalized service name and the hostname and obtaining the list of servers for the combination via DNS, along with their priorities and weights. • Top-priority SRV records (ones that share the minimum priority value) resolve into primary servers, and other records become backup servers. If `backup` is set with `server`, top-priority SRV records resolve into backup servers, and other records are ignored. • Weight influences the selection of servers by the assigned capacity: higher weights receive more requests. If set by both the `server` directive and the SRV record, the weight set by `server` is used.

This example will look up the `_http._tcp.backend.example.com` record:

```
server backend.example.com service=http resolve;
```

Added in version 1.4.0.

`slow_start=`*time*	sets the *time* to recover the `weight` for a server that goes back online, if load balancing uses the *round-robin* or *least_conn* method. If the value is set and the server is again considered available and healthy as defined by *max_fails* and *upstream_probe (PRO)*, the server will steadily recover its designated weight within the allocated timeframe. If the value isn't set, the server in a similar situation will recover its designated weight immediately.

Note

If there's only one `server` in an upstream, `slow_start` has no effect and will be ignored.

state (PRO)

Added in version 1.4.0: PRO

Syntax	`state` *file*;
Default	—
Context	upstream

Specifies the *file* where the upstream's server list is persisted. When installing from our packages, a designated `/var/lib/angie/state/` (`/var/db/angie/state/` on FreeBSD) directory with appropriate permissions is created to store these files, so you will only need to add the file's basename in the configuration:

```
upstream backend {

    zone backend 1m;
    state /var/lib/angie/state/<FILE NAME>;
}
```

The format of this server list is similar to `server`. The contents of the file change whenever there is any modification to servers in the */config/stream/upstreams/* section via the configuration API. The file is read at Angie start or configuration reload.

Caution

For the `state` directive to be used in an `upstream` block, the block should have no `server` directives; instead, it must have a shared memory zone (*zone*).

zone

Syntax	`zone` *name* [*size*];
Default	—
Context	upstream

Defines the name and size of the shared memory zone that keeps the group's configuration and run-time state that are shared between worker processes. Several groups may share the same zone. In this case, it is enough to specify the size only once.

feedback (PRO)

Added in version 1.7.0: PRO

Syntax	`feedback` *variable* [`inverse`] [`factor=`*number*] [`account=`*condition_variable*] [`last_byte`];
Default	—
Context	upstream

Enables a feedback-based load balancing mechanism for the `upstream`. It adjusts the load balancing decisions dynamically, multiplying each peer's weight by its average feedback value that is affected by the value of a *variable* over time and is subject to an optional condition.

The following parameters are accepted:

`variable`	The variable from which the feedback value is taken. It should represent a performance or health metric, and is intended to be supplied by the peer. The value is assessed at each response from the peer and factored into the rolling average according to `inverse` and `factor` settings.
`inverse`	If set, the feedback value is interpreted inversely, meaning lower values indicate better performance.
`factor`	The factor by which the feedback value is weighted when calculating the average. Valid values are integers between 0 and 99. By default — `90`. The average feedback is calculated using the exponential moving average formula. The larger is the factor, the less is the average affected by new values; if the factor is set to `90`, the result has 90% of the previous value and only 10% of the new value.
`account`	Specifies a condition variable that controls how connections are included in the calculation. The average is updated with the feedback value only if the condition variable isn't `""` or `"0"`. **Note** By default, traffic from *probes* isn't included in the calculation; combining the *$upstream_probe* variable with `account` allows to include them or even exclude everything else.

Example:

```
upstream backend {

    zone backend 1m;

    feedback $feedback_value factor=80 account=$condition_value;

    server backend1.example.com:1935  weight=1;
    server backend2.example.com:1935  weight=2;
}

map $protocol $feedback_value {
    "TCP"                     100;
    "UDP"                     75;
    default                   10;
}

map $upstream_probe $condition_value {
    "high_priority" "1";
    "low_priority"  "0";
    default         "1";
}
```

This categorizes servers into different feedback levels based on specific protocols used for different sessions, and also adds a condition mapped from *$upstream_probe* to account only for the `high_priority` probe or regular client sessions.

hash

Syntax	`hash` *key* [`consistent`];
Default	—
Context	upstream

Specifies a load balancing method for a server group where the client-server mapping is based on the hashed key value. The key can contain text, variables, and their combinations (1.11.2). Usage example:

```
hash $remote_addr;
```

Note that adding or removing a server from the group may result in remapping most of the keys to different servers. The method is compatible with the Cache::Memcached Perl library.

If the `consistent` parameter is specified, the ketama consistent hashing method will be used instead. The method ensures that only a few keys will be remapped to different servers when a server is added to or removed from the group. This helps to achieve a higher cache hit ratio for caching servers. The method is compatible with the Cache::Memcached::Fast Perl library with the ketama_points parameter set to 160.

least_conn

Syntax	`least_conn`;
Default	—
Context	upstream

Specifies that a group should use a load balancing method where a connection is passed to the server with the least number of active connections, taking into account weights of servers. If there are several such servers, they are tried in turn using a weighted round-robin balancing method.

least_time (PRO)

Syntax	`least_time` `connect` | `first_byte` | `last_byte` [`factor=`*number*] [`account=`*condition_variable*];
Default	—
Context	upstream

Sets the load balancing method for a group where the probability of forwarding a connection to an active server is inversely proportional to the average time it takes to respond; the smaller the response time, the more conections the server will receive.

`connect`	The directive accounts for the average time to establish the connection.
`first_byte`	The directive uses the average time to receive the first byte of the response.
`last_byte`	The directive uses the average time to receive the entire response.

Added in version 1.7.0: PRO

`factor`	Serves the same purpose as *response_time_factor (PRO)* and overrides it if set.
`account`	Specifies a condition variable that controls which connections should be included in the calculation. The average is updated only if the condition variable for the connection isn't `""` or `"0"`.

> **Note**
>
> By default, *probes* aren't included in the calculation; combining the *$upstream_probe* variable with `account` allows to include them or even exclude everything else.

The respective moving averages, adjusted for `factor` and `account`, are also presented as `connect_time`, `first_byte_time`, and `last_byte_time` in the `health` object of the server among the *stream upstream*

metrics in the API.

random

Syntax	`random` [`two`];
Default	—
Context	upstream

Specifies that a group should use a load balancing method where a request is passed to a randomly selected server, taking into account weights of servers.

The optional `two` parameter instructs Angie to randomly select two servers and then choose a server using the specified method. The default method is *least_conn* which passes a request to a server with the least number of active connections.

response_time_factor (PRO)

Syntax	`response_time_factor` *number*;
Default	`response_time_factor 90;`
Context	upstream

Sets the smoothing factor for the *least_time (PRO)* load balancing method, using the **previous** value when calculating the average response time according to the formula of the exponential weighted moving average.

The larger the specified *number*, the less new values influence the average; if `90` is specified, 90% of the previous value will be taken, and only 10% of the new value. Acceptable values range from 0 to 99 inclusive.

The respective moving averages are presented as `connect_time` (time to establish the connection), `first_byte_time` (time to receive the first byte of the response), and `last_byte_time` (time to receive the complete response) in the `health` object of the server among the *stream upstream metrics* in the API.

Note

Only successful responses are considered in the calculation; what constitutes an unsuccessful response is determined by the *proxy_next_upstream* directives.

sticky

Added in version 1.6.0: Angie

Added in version 1.6.0: Angie PRO

Syntax	`sticky` route *$variable...*; `sticky` learn `zone`=*zone* `create`=*$create_var1...* `lookup`=*$lookup_var1...* [`connect`] [`timeout`=*time*];
Default	—
Context	upstream

Configures the binding of client sessions to proxied servers in the mode specified by the first parameter; to drain requests from servers that have `sticky` defined, use the `drain` option in the *server* block.

Attention

The `sticky` directive must be used after all directives that set the load balancing method; otherwise, it won't work.

`route` mode

This mode uses predefined route identifiers that can be embedded in any connection properties Angie can access. It is less flexible because it relies on predefined values but can suit better if such identifiers are already in place.

Here, when a connection is established with the proxied server, it can assign a route to the client and return its identifier in a manner that they both are aware of. The value of the *sid* parameter of the *server* directive must be used as the route identifier. Note that the parameter is additionally hashed if the *sticky_secret* directive is set.

Subsequent connections from clients that wish to use this route must contain the identifier issued by the server in a way that ensures it ends up in Angie variables.

The directive lists specific variables used for routing. To select the server where the incoming connection is routed, the first non-empty variable is used; it is then compared with the *sid* parameter of the *server* directive. If selecting a server fails or the chosen server can't accept the connection, another server is selected according to the configured balancing method.

Here, Angie looks for the identifier in a custom `$route` variable, which is mapped from *$ssl_preread_server_name* (note that *ssl_preread* must be enabled):

```
stream {

    map $ssl_preread_server_name $route {

        a.example.com            a;
        b.example.com            b;
        default                  "";
    }

    upstream backend {

        server 127.0.0.1:8081 sid=a;
        server 127.0.0.1:8082 sid=b;

        sticky route $route;
    }

    server {

        listen 127.0.0.1:8080;

        ssl_preread on;

        proxy_pass backend;
    }
}
```

`learn` mode (PRO)

This mode uses a dynamically generated key to associate a client with a particular proxied server; it's more flexible because it assigns servers on the go, stores sessions in a shared memory zone, and supports different ways of passing session identifiers.

Here, a session is created based on the connection properties from the proxied server. The `create` and

`lookup` parameters list variables indicating how new sessions are created and existing sessions are looked up. Both parameters can occur multiple times.

The session identifier is the value of the first non-empty variable specified with `create`; for example, this could be the *name of the proxied server*.

Sessions are stored in a shared memory zone; its name and size are set by the `zone` parameter. If a session has been inactive for the time set by `timeout`, it is deleted. The default is 1 hour.

Subsequent connections from clients that wish to use the session must contain its identifier, ensuring that it ends up in a non-empty variable specified with `lookup`; its value will then be matched against sessions in shared memory. If selecting a server fails or the chosen server can't accept the connection, another server is selected according to the configured balancing method.

The `connect` parameter allows creating a session immediately after the connection to the proxied server was established. Without it, a session is created only after processing the connection.

In the example, Angie creates and looks up sessions, using the *$rdp_cookie* variable:

```
stream {

    upstream backend {

        server 127.0.0.1:3390 sid=a;
        server 127.0.0.1:3391 sid=b;

        sticky learn lookup=$rdp_cookie create=$rdp_cookie zone=sessions:1m;
    }

    server {

        listen 127.0.0.1:3389;

        ssl_preread on;

        proxy_pass backend;
    }
}
```

sticky_strict

Added in version 1.6.0: Angie

Added in version 1.6.0: Angie PRO

Syntax	`sticky_strict on | off;`
Default	`sticky_strict off;`
Context	upstream

When enabled, causes Angie to return a connection error to the client if the desired server is unavailable, rather than using any other available server as it would when no servers in the upstream are available.

sticky_secret

Added in version 1.6.0: Angie

Added in version 1.6.0: Angie PRO

Syntax	`sticky_secret` *string*;
Default	—
Context	upstream

Adds the *string* as the salt value to the MD5 hashing function for the *sticky* directive in the `route` mode. The *string* may contain variables, for example, *$remote_addr*:

```
upstream backend {
    server 127.0.0.1:8081 sid=a;
    server 127.0.0.1:8082 sid=b;

    sticky route $route;
    sticky_secret my_secret.$remote_addr;
}
```

Salt is appended to the value being hashed; to verify the hashing mechanism independently:

```
$ echo -n "<VALUE><SALT>" | md5sum
```

Built-in Variables

The `stream_upstream` module supports the following built-in variables:

`$upstream_addr`

stores the IP address and port, or the path to the UNIX domain socket of the upstream server. If several servers were contacted during request processing, their addresses are separated by commas, e.g.:

192.168.1.1:1935, 192.168.1.2:1935, unix:/tmp/sock

If a server cannot be selected, the variable keeps the *name* of the *server group*.

`$upstream_bytes_received`

number of bytes received from an upstream server. Values from several connections are separated by commas like addresses in the *$upstream_addr* variable.

`$upstream_bytes_sent`

number of bytes sent to an upstream server. Values from several connections are separated by commas like addresses in the *$upstream_addr* variable.

`$upstream_connect_time`

time to connect to the upstream server; the time is kept in seconds with millisecond resolution. Times of several connections are separated by commas like addresses in the *$upstream_addr* variable.

`$upstream_first_byte_time`

time to receive the first byte of data; the time is kept in seconds with millisecond resolution. Times of several connections are separated by commas like addresses in the *$upstream_addr* variable.

`$upstream_session_time_<name>`

session duration in seconds with millisecond resolution. Times of several connections are separated by commas like addresses in the *$upstream_addr* variable.

`$upstream_sticky_status`

Status of sticky connections.

`""`	Connection routed to upstream without sticky enabled.
`NEW`	Connection without sticky information.
`HIT`	Connection with sticky information routed to the desired backend.
`MISS`	Connection with sticky information routed to the backend selected by the load balancing algorithm.

Values from multiple connections are separated by commas and colons, similar to addresses in the *$upstream_addr* variable.

Upstream Probe

The module implements active health probes for *stream_upstream*.

Configuration Example

```
server {
    listen ...;

    # ...
    proxy_pass backend;
    upstream_probe_timeout 1s;

    upstream_probe backend_probe
        port=12345
        interval=5s
        test=$good
        essential
        fails=3
        passes=3
        max_response=512k
        mode=onfail
        "send=data:GET / HTTP/1.0\r\n\r\n";
}
```

Note

According to RFC 2616 (HTTP/1.1) and RFC 9110 (HTTP Semantics), HTTP headers must be separated by a CRLF sequence (`rn`) rather than just `n`.

Directives

upstream_probe (PRO)

Added in version 1.4.0: PRO

Syntax	`upstream_probe` *name* [`port=`*number*] [`interval=`*time*] [`test=`*condition*] [`essential` [`persistent`]] [`fails=`*number*] [`passes=`*number*] [`max_response=`*size*] [`mode=always` | `idle` | `onfail`] [`udp`] [`send=`*string*];
Default	—
Context	server

Defines an active health probe for peers within the *upstream* groups that are specified for *proxy_pass* in the same `location` context with the `upstream_probe` directive. Subsequently, Angie regularly probes each peer of the upstream group according to the parameters configured here.

A peer's probe is passed if the request to the peer succeeds, considering all parameter settings of the `upstream_probe` directive and the settings that control how upstreams are used by the directive's `location` context, including the *proxy_next_upstream* directive.

To make use of the probes, the upstream must have a shared memory zone (*zone*). One upstream may be configured with several probes.

The following parameters are accepted:

`name`	Mandatory name of the probe.
`port`	Alternative port number for the probe request.
`interval`	*Interval* between probes. By default — `5s`.
`test`	The condition for the probe, defined as a string of variables. If the variables' substitution yields `""` or `"0"`, the probe is not passed.
`essential`	If set, the initial state of the peer is being checked, so the peer doesn't receive client requests until the probe is passed.
`persistent`	Setting this parameter requires enabling `essential` first; `persistent` peers that were deemed healthy prior to a *configuration reload* start receiving requests without being required to pass this probe first.
`fails`	Number of subsequent failed probes that renders the peer unhealthy. By default — 1.
`passes`	Number of subsequent passed probes that renders the peer unhealthy. By default — 1.
`max_response`	Maximum memory size for the response. If a zero *value* is specified, response waiting is disabled. By default — `256k`.
`mode`	Probe mode, depending on the peers' health: • `always` — peers are probed regardless of their state; • `idle` — probes affect unhealthy peers and peers where `interval` has elapsed since the last client request. • `onfail` — only unhealthy peers are probed. By default — `always`.
`udp`	If specified, the UDP protocol is used for probing. By default, TCP is used for probing.
`send`	Data sent for the check; this can be a string with the prefix `data:` or a file name with data (specified absolutely or relative to the `/usr/local/angie/` directory).

Example:

```
upstream backend {
    zone backend 1m;

    server a.example.com;
    server b.example.com;
}

map $upstream_probe_response $good {
    ~200    "1";
    default  "";
}

server {
    listen ...;

    # ...
    proxy_pass backend;
```

```
    upstream_probe_timeout 1s;

    upstream_probe backend_probe
        port=12345
        interval=5s
        test=$good
        essential
        persistent
        fails=3
        passes=3
        max_response=512k
        mode=onfail
        "send=data:GET / HTTP/1.0\r\n\r\n";
}
```

Details of probe operation:

- Initially, the peer won't receive client requests until it passes *all* `essential` probes configured for it, skipping `persistent` ones if the configuration was reloaded and the peer was deemed healthy prior to that. If there are no such probes, the peer is considered healthy.
- The peer is considered unhealthy and won't receive client requests, if *any* of the probes configured for it hits `fails` or the peer reaches *max_fails*.
- For an unhealthy peer to be considered healthy again, *all* probes configured for it must reach their respective `passes`; after that, *max_fails* is also considered.

Built-in Variables

The `stream_upstream` module supports the following built-in variables:

`$upstream_probe` (PRO)

Name of the currently active *upstream_probe*.

`$upstream_probe_response` (PRO)

Contents of the response received during an active probe configured by *upstream_probe*.

The core stream module implements basic functionality for handling TCP and UDP connections: this includes defining server blocks, traffic routing, configuring proxying, SSL/TLS support, and managing connections for streaming services, such as databases, DNS, and other protocols that operate over TCP and UDP.

The other modules in this section extend this functionality, allowing you to flexibly configure and optimize the stream server for various scenarios and requirements.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-stream` build option. In packages and images from our repos, the module is included in the build.

Configuration Example

```
worker_processes auto;

error_log /var/log/angie/error.log info;

events {
    worker_connections  1024;
}
```

```
stream {
    upstream backend {
        hash $remote_addr consistent;

        server backend1.example.com:12345 weight=5;
        server 127.0.0.1:12345            max_fails=3 fail_timeout=30s;
        server unix:/tmp/backend3;
    }

    upstream dns {
       server 192.168.0.1:53535;
       server dns.example.com:53;
    }

    server {
        listen 12345;
        proxy_connect_timeout 1s;
        proxy_timeout 3s;
        proxy_pass backend;
    }

    server {
        listen 127.0.0.1:53 udp reuseport;
        proxy_timeout 20s;
        proxy_pass dns;
    }

    server {
        listen [::1]:12345;
        proxy_pass unix:/tmp/stream.socket;
    }
}
```

Directives

listen

Syntax	`listen` *address*[:*port*] [`ssl`] [`udp`] [`proxy_protocol`] [`setfib=`*number*] [`fastopen=`*number*] [`backlog=`*number*] [`rcvbuf=`*size*] [`sndbuf=`*size*] [`accept_filter=`*filter*] [`deferred`] [`bind`] [`ipv6only=on` | `off`] [`reuseport`] [`so_keepalive=`on|off|[*keepidle*]:[*keepintvl*]:[*keepcnt*]];
Default	—
Context	server

Sets the *address* and *port* for the socket on which the server will accept connections. It is possible to specify just the *port*, so Angie listens on all available IPv4 (and IPv6, if enabled) interfaces. The address can also be a hostname, for example:

```
listen 127.0.0.1:12345;
listen *:12345;
listen 12345;      # same as *:12345
listen localhost:12345;
```

IPv6 addresses are specified in square brackets:

```
listen [::1]:12345;
listen [::]:12345;
```

UNIX domain sockets are specified with the `unix:` prefix:

```
listen unix:/var/run/angie.sock;
```

Port ranges are specified with the first and last port separated by a hyphen:

```
listen 127.0.0.1:12345-12399;
listen 12345-12399;
```

Important

Different servers must listen on different *address:port* pairs.

`ssl`	allows specifying that all connections accepted on this port should work in SSL mode.
`udp`	configures a listening socket for working with datagrams. In order to handle packets from the same address and port in the same session, the *reuseport* parameter should also be specified.
`proxy_protocol`	allows specifying that all connections accepted on this port should use the PROXY protocol.

The `listen` directive can have several additional parameters specific to socket-related system calls.

`setfib=`*number*	sets the associated routing table, FIB (the `SO_SETFIB` option) for the listening socket. This currently works only on FreeBSD.
`fastopen=`*number*	enables "TCP Fast Open" for the listening socket and limits the maximum length for the queue of connections that have not yet completed the three-way handshake.

Caution

Do not enable this feature unless the server can handle receiving the same SYN packet with data more than once.

`backlog=`*number*	sets the `backlog` parameter in the `listen()` call that limits the maximum length for the queue of pending connections. By default, `backlog` is set to `-1` on FreeBSD, DragonFly BSD, and macOS, and to 511 on other platforms.
`rcvbuf=`*size*	sets the receive buffer size (the `SO_RCVBUF` option) for the listening socket.
`sndbuf=`*size*	sets the send buffer size (the `SO_SNDBUF` option) for the listening socket.
`accept_filter=`*filt*	Sets the name of accept filter (the `SO_ACCEPTFILTER` option) for the listening socket that filters incoming connections before passing them to `accept()`. This works only on FreeBSD and NetBSD 5.0+. Acceptable values are `dataready` and `httpready`.
`deferred`	instructs to use a deferred `accept()` (the `TCP_DEFER_ACCEPT` socket option) on Linux.
`bind`	this parameter instructs to make a separate `bind()` call for a given *address:port* pair. The fact is that if there are several *listen* directives with the same *port* but different addresses, and one of the `listen` directives listens on all addresses for the given port (**:port*), Angie will `bind()` only to **:port*. It should be noted that the `getsockname()` system call will be made in this case to determine the address that accepted the connection. If the `setfib`, `fastopen`, `backlog`, `rcvbuf`, `sndbuf`, `accept_filter`, `deferred`, `ipv6only`, `reuseport`, or `so_keepalive` parameters are used then for a given *address:port* pair a separate `bind()` call will always be made.
`ipv6only=on` | `off`	this parameter determines (via the `IPV6_V6ONLY` socket option) whether an IPv6 socket listening on a wildcard address *[::]* will accept only IPv6 connections or both IPv6 and IPv4 connections. This parameter is turned on by default. It can only be set once on start.
`reuseport`	this parameter instructs to create an individual listening socket for each worker process (using the `SO_REUSEPORT` socket option on Linux 3.9+ and DragonFly BSD, or `SO_REUSEPORT_LB` on FreeBSD 12+), allowing a kernel to distribute incoming connections between worker processes. This currently works only on Linux 3.9+, DragonFly BSD, and FreeBSD 12+.

Caution

Inappropriate use of this option may have its security implications.

`so_keepalive=on` | `off` | [`keepidle`]:[*keepintvl*]:[*keepcnt*]` configures the "TCP keepalive" behavior for the listening socket.

`''`	if this parameter is omitted then the operating system's settings will be in effect for the socket
`on`	the *SO_KEEPALIVE* option is turned on for the socket
`off`	the *SO_KEEPALIVE* option is turned off for the socket

Some operating systems support setting of TCP keepalive parameters on a per-socket basis using the `TCP_KEEPIDLE`, `TCP_KEEPINTVL`, and `TCP_KEEPCNT` socket options. On such systems (currently, Linux 2.4+, NetBSD 5+, and FreeBSD 9.0-STABLE), they can be configured using the keepidle, keepintvl, and keepcnt parameters. One or two parameters may be omitted, in which case the system default setting for the corresponding socket option will be in effect.

For example,

```
so_keepalive=30m::10
```

will set the idle timeout (*TCP_KEEPIDLE*) to 30 minutes, leave the probe interval (*TCP_KEEPINTVL*) at its system default, and set the probes count (*TCP_KEEPCNT*) to 10 probes.

preread_buffer_size

Syntax	`preread_buffer_size` *size*;
Default	`preread_buffer_size 16k;`
Context	stream, server

Specifies a size of the *preread* buffer.

preread_timeout

Syntax	`preread_timeout` *timeout*;
Default	`preread_timeout 30s;`
Context	stream, server

Specifies a timeout of the *preread* phase.

proxy_protocol_timeout

Syntax	`proxy_protocol_timeout` *timeout*;
Default	`proxy_protocol_timeout 30s;`
Context	stream, server

Specifies a *timeout* for reading the PROXY protocol header to complete. If no entire header is transmitted within this time, the connection is closed.

resolver

Syntax	`resolver` *address* ... [`valid=`*time*] [`ipv4=on` | `off`] [`ipv6=on` | `off`] [`status_zone=`*zone*];
Default	—
Context	stream, server, upstream

Configures name servers used to resolve names of upstream servers into addresses, for example:

```
resolver 127.0.0.53 [::1]:5353;
```

The address can be specified as a domain name or IP address, with an optional port. If port is not specified, the port 53 is used. Name servers are queried in a round-robin fashion.

By default, Angie caches answers using the TTL value of a response. The optional valid parameter allows overriding it:

`valid`	*optional* valid parameter allows overriding cached entry validity

```
resolver 127.0.0.53 [::1]:5353 valid=30s;
```

By default, Angie will look up both IPv4 and IPv6 addresses while resolving.

`ipv4=off`	disables looking up of IPv4 addresses
`ipv6=off`	disables looking up of IPv6 addresses

`status_zone`	*optional* parameter; enables the collection of DNS server request and response metrics (*/status/resolvers/<zone>*) in the specified zone.

Tip

To prevent DNS spoofing, it is recommended configuring DNS servers in a properly secured trusted local network.

Tip

When running in Docker, use its internal DNS server address such as `127.0.0.11`.

resolver_timeout

Syntax	`resolver_timeout` *time;*
Default	`resolver_timeout 30s;`
Context	stream, server, upstream

Sets a timeout for name resolution, for example:

```
resolver_timeout 5s;
```

server

Syntax	`server { ... }`
Default	—
Context	stream

Sets the configuration for a server.

server_name

Syntax	`server_name` *name ...;*
Default	`server_name "";`
Context	`server`

Sets names of a virtual server.

Attention

In the `stream` module, `server_name` relies on Server Name Indication (*SNI*) and only works with TLS connections. To use it, you must *configure TLS termination* or *enable TLS preread* in the corresponding `server` block.

Example configuration:

```
server {
    listen 443 ssl;
    server_name example.com www.example.com;
    ssl_certificate /etc/angie/cert.pem;
    ssl_certificate_key /etc/angie/key.pem;
}
```

The first name becomes the primary server name.

Server names can include an asterisk (*) to replace the first or last part of a name:

```
server {
    server_name example.com *.example.com www.example.*;
}
```

These names are called wildcard names.

You can also use regular expressions in server names by preceding the name with a tilde (~):

```
server {
    server_name www.example.com ~^www\d+\.example\.com$;
}
```

Regular expressions may include captures that can be used in other directives:

```
server {
    server_name ~^(www\.)?(.+)$;

    proxy_pass www.$2:12345;
}
```

Named captures in regular expressions create variables that can be used in other directives:

```
server {
    server_name ~^(www\.)?(?<domain>.+)$;

    proxy_pass www.$domain:12345;
}
```

If the directive's parameter is set to `$hostname`, the machine's hostname is inserted.

When searching for a virtual server by name, if the name matches more than one of the specified variants (e.g., both a wildcard name and a regular expression match), the first matching variant will be chosen in the following order of priority:

- The exact name
- The longest wildcard name starting with an asterisk, e.g., `*.example.com`
- The longest wildcard name ending with an asterisk, e.g., `mail.*`
- The first matching regular expression (in order of appearance in the configuration file)

server_names_hash_bucket_size

Syntax	`server_names_hash_bucket_size` *size*;
Default	`server_names_hash_bucket_size 32|64|128;`
Context	stream

Sets the bucket size for the server names hash tables. The default value depends on the size of the processor's cache line.

server_names_hash_max_size

Syntax	`server_names_hash_max_size` *size*;
Default	`server_names_hash_max_size 512;`
Context	stream

Sets the maximum size of the server names hash tables.

status_zone

Syntax	`status_zone` *zone* | *key* `zone=`*zone*[:*count*];
Default	—
Context	*server*

Allocates a shared memory zone to collect metrics for */status/stream/server_zones/<zone>*.

Multiple `server` contexts can share the same zone for data collection.

The single-value *zone* syntax aggregates all metrics for its context in the same shared memory zone:

```
server {

    listen 80;
    server_name *.example.com;

    status_zone single;
    # ...
}
```

The alternative syntax uses the folowing parameters:

key	A string with variables, whose value determines the grouping of connections in the zone. All connections producing identical values after substitution are grouped together. If substitution yields an empty value, metrics aren't updated.
zone	The name of the shared memory zone.
count (optional)	The maximum number of separate groups for collecting metrics. If new *key* values would exceed this limit, they are grouped under *zone* instead. The default value is 1.

In the following example, all connections sharing the same `$server_addr` value are grouped into the `host_zone`. Metrics are tracked separately for each unique `$server_addr` until there are 10 metric groups. Once this limit is reached, any additional `$server_addr` values are included under the `server_zone`:

```
stream {

    upstream backend {
        server 192.168.0.1:3306;
        server 192.168.0.2:3306;
        # ...
    }

    server {

        listen 3306;
        proxy_pass backend;
```

```
        status_zone $server_addr zone=server_zone:10;
    }
}
```

The resulting metrics are thus split between individual servers in the API output.

stream

Syntax	`stream { ... }`
Default	—
Context	main

Provides the configuration file context in which the stream server directives are specified.

tcp_nodelay

Syntax	`tcp_nodelay on | off;`
Default	`tcp_nodelay on;`
Context	stream, server

Enables or disables the use of the *TCP_NODELAY* option. The option is enabled for both client and proxied server connections.

variables_hash_bucket_size

Syntax	`variables_hash_bucket_size` *size*;
Default	`variables_hash_bucket_size 64;`
Context	stream

Sets the bucket size for the variables hash table. The details of setting up hash tables are provided in a separate *document*.

variables_hash_max_size

Syntax	`variables_hash_max_size` *size*;
Default	`variables_hash_max_size 1024;`
Context	stream

Sets the maximum size of the variables hash table. The details of setting up hash tables are provided in a separate *document*.

Built-in Variables

The *stream core* module supports following variables:

`$angie_version`

Angie version

`$binary_remote_addr`

client address in a binary form, value's length is always 4 bytes for IPv4 addresses or 16 bytes for IPv6 addresses

`$bytes_received`

number of bytes received from a client

`$bytes_sent`

number of bytes sent to a client

`$connection`

connection serial number

`$hostname`

host name

`$msec`

current time in seconds with the milliseconds resolution

`$pid`

PID of the worker process

`$protocol`

protocol used to communicate with the client: `TCP` or `UDP`

`$proxy_protocol_addr`

client address from the PROXY protocol header The PROXY protocol must be previously enabled by setting the *proxy_protocol* parameter in the *listen* directive.

`$proxy_protocol_port`

client port from the PROXY protocol header The PROXY protocol must be previously enabled by setting the *proxy_protocol* parameter in the *listen* directive.

`$proxy_protocol_server_addr`

server address from the PROXY protocol header The PROXY protocol must be previously enabled by setting the *proxy_protocol* parameter in the *listen* directive.

`$proxy_protocol_server_port`

server port from the PROXY protocol header The PROXY protocol must be previously enabled by setting the *proxy_protocol* parameter in the *listen* directive.

`$proxy_protocol_tlv_<name>`

TLV from the PROXY Protocol header. The *name* can be a TLV type or its numeric value. In the latter case, the value is hexadecimal and should be prefixed with *0x*:

$proxy_protocol_tlv_alpn $proxy_protocol_tlv_0x01

SSL TLVs can also be accessed by TLV type name or its numeric value, both prefixed by *ssl_*:

$proxy_protocol_tlv_ssl_version $proxy_protocol_tlv_ssl_0x21

The following TLV type names are supported:

- *alpn (0x01)* - upper layer protocol used over the connection
- *authority (0x02)* - host name value passed by the client
- *unique_id (0x05)* - unique connection id
- *netns (0x30)* - name of the namespace
- *ssl (0x20)* - binary SSL TLV structure

The following SSL TLV type names are supported:

- *ssl_version (0x21)* - SSL version used in client connection
- *ssl_cn (0x22)* - SSL certificate Common Name
- *ssl_cipher (0x23)* - name of the used cipher
- *ssl_sig_alg (0x24)* - algorithm used to sign the certificate
- *ssl_key_alg (0x25)* - public-key algorithm

Also, the following special SSL TLV type name is supported:

- *ssl_verify* - client SSL certificate verification result, *0* if the client presented a certificate and it was successfully verified, non-zero otherwise.

The PROXY protocol must be previously enabled by setting the *proxy_protocol* parameter in the *listen* directive.

`$remote_addr`

client address

`$remote_port`

client port

`$server_addr`

an address of the server which accepted a connection Computing a value of this variable usually requires one system call. To avoid a system call, the *listen* directives must specify addresses and use the `bind` parameter.

`$server_port`

port of the server which accepted a connection

`$session_time`

session duration in seconds with a milliseconds resolution

`$status`

session status, can be one of the following:

200	session completed successfully
400	client data could not be parsed, for example, the PROXY protocol header
403	access forbidden, for example, when access is limited for *certain client addresses*
500	internal server error
502	bad gateway, for example, if an upstream server could not be selected or reached
503	service unavailable, for example, when access is limited by the *number of connections*

`$time_iso8601`

local time in the ISO 8601 standard format

`$time_local`

local time in the Common Log Format

Mail Module

Auth HTTP

The module enables subrequest-based authentication by sending an additional HTTP request before processing the main request. If the subrequest returns a 2xx status, the main request proceeds; if it returns 401 or 403, the appropriate error is sent to the user, while any other response triggers a 500 error. This approach is typically used to delegate authentication to external services, unify authentication across applications, or integrate with third-party systems like OAuth or LDAP.

Directives

auth_http

Syntax	`auth_http` *uri*;
Default	—
Context	mail, server

Sets the URL of the HTTP authentication server. The protocol is described *below*.

auth_http_header

Syntax	`auth_http_header` *header value*;
Default	—
Context	mail, server

Appends the specified header to requests sent to the authentication server. This header can be used as the shared secret to verify that the request comes from Angie. For example:

```
auth_http_header X-Auth-Key "secret_string";
```

auth_http_pass_client_cert

Syntax	`auth_http_pass_client_cert on | off;`
Default	`auth_http_pass_client_cert off;`
Context	mail, server

Appends the "Auth-SSL-Cert" header with the *client* certificate in the PEM format (urlencoded) to requests sent to the authentication server.

auth_http_timeout

Syntax	`auth_http_timeout` *time*;
Default	`auth_http_timeout 60s;`
Context	mail, server

Sets the timeout for communication with the authentication server.

Protocol

The HTTP protocol is used to communicate with the authentication server. The data in the response body is ignored, the information is passed only in the headers.

Examples of requests and responses:

Request:

```
GET /auth HTTP/1.0
Host: localhost
Auth-Method: plain # plain/apop/cram-md5/external
Auth-User: user
Auth-Pass: password
Auth-Protocol: imap # imap/pop3/smtp
Auth-Login-Attempt: 1
Client-IP: 192.0.2.42
Client-Host: client.example.org
```

Good response:

```
HTTP/1.0 200 OK
Auth-Status: OK
Auth-Server: 198.51.100.1
Auth-Port: 143
```

Bad response:

```
HTTP/1.0 200 OK
Auth-Status: Invalid login or password
Auth-Wait: 3
```

If there is no "Auth-Wait" header, an error will be returned and the connection will be closed. The current implementation allocates memory for each authentication attempt. The memory is freed only at the end of a session. Therefore, the number of invalid authentication attempts in a single session must be limited — the server must respond without the "Auth-Wait" header after 10-20 attempts (the attempt number is passed in the "Auth-Login-Attempt" header).

When the APOP or CRAM-MD5 are used, request-response will look as follows:

```
GET /auth HTTP/1.0
Host: localhost
Auth-Method: apop
Auth-User: user
Auth-Salt: <238188073.1163692009@mail.example.com>
Auth-Pass: auth_response
Auth-Protocol: imap
```

```
Auth-Login-Attempt: 1
Client-IP: 192.0.2.42
Client-Host: client.example.org
```

Good response:

```
HTTP/1.0 200 OK
Auth-Status: OK
Auth-Server: 198.51.100.1
Auth-Port: 143
Auth-Pass: plain-text-pass
```

If the "Auth-User" header exists in the response, it overrides the username used to authenticate with the backend.

For the SMTP, the response additionally takes into account the "Auth-Error-Code" header — if exists, it is used as a response code in case of an error. Otherwise, the *535 5.7.0* code will be added to the "Auth-Status" header.

For example, if the following response is received from the authentication server:

```
HTTP/1.0 200 OK
Auth-Status: Temporary server problem, try again later
Auth-Error-Code: 451 4.3.0
Auth-Wait: 3
```

then the SMTP client will receive an error

```
451 4.3.0 Temporary server problem, try again later
```

If proxying SMTP does not require authentication, the request will look as follows:

```
GET /auth HTTP/1.0
Host: localhost
Auth-Method: none
Auth-User:
Auth-Pass:
Auth-Protocol: smtp
Auth-Login-Attempt: 1
Client-IP: 192.0.2.42
Client-Host: client.example.org
Auth-SMTP-Helo: client.example.org
Auth-SMTP-From: MAIL FROM: <>
Auth-SMTP-To: RCPT TO: <postmaster@mail.example.com>
```

For the SSL/TLS client connection, the "Auth-SSL" header is added, and "Auth-SSL-Verify" will contain the result of client certificate verification, if *enabled*: `SUCCESS`, `FAILED:reason`, and `NONE` if a certificate was not present.

When the client certificate was present, its details are passed in the following request headers: "Auth-SSL-Subject", "Auth-SSL-Issuer", "Auth-SSL-Serial", and "Auth-SSL-Fingerprint". If *auth_http_pass_client_cert* is enabled, the certificate itself is passed in the "Auth-SSL-Cert" header. The protocol and cipher of the established connection are passed in the "Auth-SSL-Protocol" and "Auth-SSL-Cipher" headers. The request will look as follows:

```
GET /auth HTTP/1.0
Host: localhost
Auth-Method: plain
Auth-User: user
Auth-Pass: password
```

```
Auth-Protocol: imap
Auth-Login-Attempt: 1
Client-IP: 192.0.2.42
Auth-SSL: on
Auth-SSL-Protocol: TLSv1.3
Auth-SSL-Cipher: TLS_AES_256_GCM_SHA384
Auth-SSL-Verify: SUCCESS
Auth-SSL-Subject: /CN=example.com
Auth-SSL-Issuer: /CN=example.com
Auth-SSL-Serial: C07AD56B846B5BFF
Auth-SSL-Fingerprint: 29d6a80a123d13355ed16b4b04605e29cb55a5ad
```

When the *PROXY protocol* is used, its details are passed in the following request headers: "Proxy-Protocol-Addr", "Proxy-Protocol-Port", "Proxy-Protocol-Server-Addr", and "Proxy-Protocol-Server-Port".

IMAP

The module enables IMAP mail protocol support, allowing the server to interact with mail storage systems. It establishes connections to IMAP servers, processes common commands like listing mailboxes and retrieving messages, and provides secure authentication while managing message statuses.

Directives

imap_auth

Syntax	`imap_auth` *method* ...;
Default	`imap_auth plain;`
Context	mail, server

Sets permitted methods of authentication for IMAP clients. Supported methods are:

`plain`	LOGIN, AUTH=PLAIN
`login`	AUTH=LOGIN
`cram-md5`	AUTH=CRAM-MD5. In order for this method to work, the password must be stored unencrypted.
`external`	AUTH=EXTERNAL

Plain text authentication methods (the `LOGIN` command, `AUTH=PLAIN`, and `AUTH=LOGIN`) are always enabled, though if the plain and login methods are not specified, `AUTH=PLAIN` and `AUTH=LOGIN` will not be automatically included in *imap_capabilities*.

imap_capabilities

Syntax	`imap_capabilities` *extension* ...;
Default	`imap_capabilities IMAP4 IMAP4rev1 UIDPLUS;`
Context	mail, server

Sets the IMAP protocol extensions list that is passed to the client in response to the CAPABILITY command. The authentication methods specified in the *imap_auth* directive and STARTTLS are automatically added to this list depending on the *starttls* directive value.

It makes sense to specify the extensions supported by the IMAP backends to which the clients are proxied (if these extensions are related to commands used after the authentication, when Angie transparently proxies a client connection to the backend).

imap_client_buffer

Syntax	`imap_client_buffer` *size*;
Default	`imap_client_buffer 4k|8k;`
Context	mail, server

Sets the size of the buffer used for reading IMAP commands. By default, the buffer size is equal to one memory page. This is either 4K or 8K, depending on a platform.

POP3

The module enables POP3 mail protocol support, allowing the server to download messages from mail servers. It connects to POP3 servers, retrieves message headers and content, provides secure authentication, and manages message statuses like downloaded or deleted.

Directives

pop3_auth

Syntax	`pop3_auth` *method* ...;
Default	`pop3_auth plain;`
Context	mail, server

Sets permitted methods of authentication for POP3 clients. Supported methods are:

`plain`	USER/PASS, AUTH PLAIN, AUTH LOGIN
`apop`	APOP. In order for this method to work, the password must be stored unencrypted.
`cram-md5`	AUTH=CRAM-MD5. In order for this method to work, the password must be stored unencrypted.
`external`	AUTH=EXTERNAL

Plain text authentication methods (`USER/PASS`, `AUTH PLAIN` and `AUTH LOGIN`) are always enabled, though if the `plain` method is not specified, `AUTH PLAIN` and `AUTH LOGIN` will not be automatically included in *pop3_capabilities*.

pop3_capabilities

Syntax	`pop3_capabilities` *extension* ...;
Default	`pop3_capabilities TOP USER UIDL;`
Context	mail, server

Sets the POP3 protocol extensions list that is passed to the client in response to the CAPA command. The authentication methods specified in the *pop3_auth* directive (SASL extension) and STLS are automatically added to this list depending on the *starttls* directive value.

It makes sense to specify the extensions supported by the POP3 backends to which the clients are proxied (if these extensions are related to commands used after the authentication, when Angie transparently proxies the client connection to the backend).

Proxy

The module enables support for mail protocols (POP3, IMAP, SMTP), allowing the server to act as a proxy between clients and backend mail servers. It establishes connections to backend servers, handles secure authentication with plain text, SSL/TLS, or STARTTLS, routes client traffic appropriately, and supports flexible authentication and server selection.

Directives

proxy_buffer

Syntax	`proxy_buffer` *size*;
Default	`proxy_buffer 4k|8k;`
Context	mail, server

Sets the size of the buffer used for proxying. By default, the buffer size is equal to one memory page. Depending on a platform, it is either 4K or 8K.

proxy_pass_error_message

Syntax	`proxy_pass_error_message on | off;`
Default	`proxy_pass_error_message off;`
Context	mail, server

Indicates whether to pass the error message obtained during the authentication on the backend to the client.

Usually, if the authentication in Angie is a success, the backend cannot return an error. If it nevertheless returns an error, it means some internal error has occurred. In such case the backend message can contain information that should not be shown to the client. However, responding with an error for the correct password is a normal behavior for some POP3 servers. The directive should be enabled in this case.

proxy_protocol

Syntax	`proxy_protocol on | off;`
Default	`proxy_protocol off;`
Context	mail, server

Enables the PROXY protocol for connections to a backend.

proxy_smtp_auth

Syntax	`proxy_smtp_auth on | off;`
Default	`proxy_smtp_auth off;`
Context	mail, server

Enables or disables user authentication on the SMTP backend using the *AUTH* command.

If *XCLIENT* is also enabled, then the *XCLIENT* command will not send the *LOGIN* parameter.

proxy_timeout

Syntax	`proxy_timeout` *time*;
Default	`proxy_timeout 24h;`
Context	mail, server

Sets the timeout between two successive read or write operations on client or proxied server connections. If no data is transmitted within this time, the connection is closed.

xclient

Syntax	`xclient on` | `off`;
Default	`xclient on;`
Context	mail, server

Enables or disables the passing of the XCLIENT command with client parameters when connecting to the SMTP backend.

With `XCLIENT`, the MTA is able to write client information to the log and apply various limitations based on this data.

If `XCLIENT` is enabled then Angie passes the following commands when connecting to the backend:

- `EHLO` with the *server name*
- `XCLIENT`
- `EHLO` or `HELO`, as passed by the client

If the name *found* by the client IP address points to the same address, it is passed in the `NAME` parameter of the `XCLIENT` command. If the name could not be found, points to a different address, or *resolver* is not specified, then `[UNAVAILABLE]` is passed in the `NAME` parameter. If an error has occurred in the process of resolving, the `[TEMPUNAVAIL]` value is used.

If `XCLIENT` is disabled, Angie passes the `EHLO` command with the *server name* when connecting to the backend if the client has passed `EHLO`, or `HELO` with the server name, otherwise.

RealIP

The module is used to change the client address and port to the ones sent in the PROXY protocol header. The PROXY protocol must be previously enabled by setting the `proxy_protocol` parameter in the *listen* directive.

Configuration Example

```
listen 110 proxy_protocol;

set_real_ip_from  192.168.1.0/24;
set_real_ip_from  192.168.2.1;
set_real_ip_from  2001:0db8::/32;
```

Directives

set_real_ip_from

Syntax	`set_real_ip_from` *address* | *CIDR* | `unix:`;
Default	—
Context	mail, server

Defines trusted addresses that are known to send correct replacement addresses. If the special value `unix:` is specified, all UNIX domain sockets will be trusted.

SMTP

The module enables support for the SMTP mail protocol, allowing the server to proxy outgoing email traffic between clients and backend mail servers. It establishes connections to SMTP servers, supports secure authentication with LOGIN or PLAIN methods, provides STARTTLS and SSL/TLS encryption, and routes client requests based on authentication results.

Directives

smtp_auth

Syntax	`smtp_auth` *method* ...;
Default	`smtp_auth plain login;`
Context	mail, server

Sets permitted methods of SASL authentication for SMTP clients. Supported methods are:

`plain`	AUTH PLAIN
`login`	AUTH LOGIN
`cram-md5`	AUTH CRAM-MD5. In order for this method to work, the password must be stored unencrypted.
`external`	AUTH EXTERNAL
`none`	Authentication is not required

Plain text authentication methods (`AUTH PLAIN` and `AUTH LOGIN`) are always enabled, though if the plain and login methods are not specified, `AUTH PLAIN` and `AUTH LOGIN` will not be automatically included in *smtp_capabilities*.

smtp_capabilities

Syntax	`smtp_capabilities` *extension* ...;
Default	—
Context	mail, server

Sets the SMTP protocol extensions list that is passed to the client in response to the EHLO command. The authentication methods specified in the *smtp_auth* directive and STARTTLS are automatically added to this list depending on the *starttls* directive value.

It makes sense to specify the extensions supported by the MTA to which the clients are proxied (if these extensions are related to commands used after the authentication, when Angie transparently proxies the client connection to the backend).

smtp_client_buffer

Syntax	`smtp_client_buffer` *size*;
Default	`smtp_client_buffer 4k|8k;`
Context	mail, server

Sets the size of the buffer used for reading SMTP commands. By default, the buffer size is equal to one memory page. This is either 4K or 8K, depending on a platform.

smtp_greeting_delay

Syntax	`smtp_greeting_delay` *time*;
Default	`smtp_greeting_delay 0;`
Context	mail, server

Allows setting a delay before sending an SMTP greeting in order to reject clients who fail to wait for the greeting before sending SMTP commands.

SSL

The module enables SSL/TLS encryption support for mail proxy protocols (POP3, IMAP, SMTP), allowing secure communication between clients and the server. It provides SSL/TLS encryption for incoming connections, supports STARTTLS upgrades, manages certificates and keys, and controls SSL settings such as ciphers and protocol versions.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-mail_ssl_module` build option.

In packages and images from our repos, the module is included in the build.

Important

This module requires the OpenSSL library.

Configuration Example

To reduce the processor load it is recommended to

- set the number of *worker processes* equal to the number of processors,
- enable the *shared* session cache,
- disable the *built-in* session cache,
- and possibly increase the session *lifetime* (by default, 5 minutes):

```
worker_processes auto;

mail {

    ...

    server {
        listen              993 ssl;

        ssl_protocols       TLSv1.2 TLSv1.3;
        ssl_ciphers         AES128-SHA:AES256-SHA:RC4-SHA:DES-CBC3-SHA:RC4-MD5;
        ssl_certificate     /usr/local/angie/conf/cert.pem;
        ssl_certificate_key /usr/local/angie/conf/cert.key;
        ssl_session_cache   shared:SSL:10m;
        ssl_session_timeout 10m;

    #    ...
    }
```

Directives

ssl_certificate

Syntax	`ssl_certificate` *file*;
Default	—
Context	mail, server

Specifies a file with the certificate in the PEM format for the given server. If intermediate certificates should be specified in addition to a primary certificate, they should be specified in the same file in the following order: the primary certificate comes first, then the intermediate certificates. A secret key in the PEM format may be placed in the same file.

This directive can be specified multiple times to load certificates of different types, for example, RSA and ECDSA:

```
server {
    listen              993 ssl;

    ssl_certificate     example.com.rsa.crt;
    ssl_certificate_key example.com.rsa.key;

    ssl_certificate     example.com.ecdsa.crt;
    ssl_certificate_key example.com.ecdsa.key;

#    ...
}
```

Only OpenSSL 1.0.2 or higher supports separate certificate chains for different certificates. With older versions, only one certificate chain can be used.

The value "data:*certificate*" can be specified instead of the `file`, which loads a certificate without using intermediate files.

Note that inappropriate use of this syntax may have its security implications, such as writing secret key data to *error log*.

ssl_certificate_key

Syntax	`ssl_certificate_key` *file*;
Default	—
Context	mail, server

Specifies a file with the secret key in the PEM format for the given server.

The value "engine:*name*:id" can be specified instead of the `file`, which loads a secret key with a specified *id* from the OpenSSL engine *name*.

The value "data:*key*" can be specified instead of the `file`, which loads a secret key without using intermediate files. Note that inappropriate use of this syntax may have its security implications, such as writing secret key data to *error log*.

ssl_ciphers

Syntax	`ssl_ciphers` *ciphers*;
Default	`ssl_ciphers HIGH:!aNULL:!MD5;`
Context	mail, server

Specifies the enabled ciphers. The ciphers are specified in the format understood by the OpenSSL library, for example:

```
ssl_ciphers ALL:!aNULL:!EXPORT56:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP;
```

The list of ciphers depends on the version of OpenSSL installed. The full list can be viewed using the `openssl ciphers` command.

Attention

The `ssl_ciphers` directive does *not* configure ciphers for TLS 1.3 when using OpenSSL. To tune TLS 1.3 ciphers with OpenSSL, use the *ssl_conf_command* directive, which was added to support advanced SSL configuration.

- In LibreSSL, TLS 1.3 ciphers *can* be configured using `ssl_ciphers`.
- In BoringSSL, TLS 1.3 ciphers cannot be configured at all.

ssl_client_certificate

Syntax	`ssl_client_certificate` *file*;
Default	—
Context	mail, server

Specifies a file with trusted CA certificates in the PEM format used to *verify* client certificates.

The list of certificates will be sent to clients. If this is not desired, the *ssl_trusted_certificate* directive can be used.

ssl_conf_command

Syntax	`ssl_conf_command` *name value*;
Default	—
Context	mail, server

Sets arbitrary OpenSSL configuration commands.

Important

The directive is supported when using OpenSSL 1.0.2 or higher. To configure TLS 1.3 ciphers with OpenSSL, use the `ciphersuites` command.

Several *ssl_conf_command* directives can be specified on the same level:

```
ssl_conf_command Options PrioritizeChaCha;
ssl_conf_command Ciphersuites TLS_CHACHA20_POLY1305_SHA256;
```

These directives are inherited from the previous configuration level if and only if there are no *ssl_conf_command* directives defined on the current level.

Caution

Note that configuring OpenSSL directly might result in unexpected behavior.

ssl_crl

Syntax	`ssl_crl` *file*;
Default	—
Context	mail, server

Specifies a file with revoked certificates (CRL) in the PEM format used to *verify* client certificates.

ssl_dhparam

Syntax	`ssl_dhparam` *file*;
Default	—
Context	mail, server

Specifies a file with DH parameters for DHE ciphers.

Caution

By default no parameters are set, and therefore DHE ciphers will not be used.

ssl_ecdh_curve

Syntax	`ssl_ecdh_curve` *curve*;
Default	`ssl_ecdh_curve auto;`
Context	mail, server

Specifies a curve for ECDHE ciphers.

Important

When using OpenSSL 1.0.2 or higher, it is possible to specify multiple curves, for example:

```
ssl_ecdh_curve prime256v1:secp384r1;
```

The special value `auto` instructs Angie to use a list built into the OpenSSL library when using OpenSSL 1.0.2 or higher, or prime256v1 with older versions.

Important

When using OpenSSL 1.0.2 or higher, this directive sets the list of curves supported by the server. Thus, in order for ECDSA certificates to work, it is important to include the curves used in the certificates.

ssl_password_file

Syntax	`ssl_password_file` *file*;
Default	—
Context	mail, server

Specifies a file with passphrases for *secret keys* where each passphrase is specified on a separate line. Passphrases are tried in turn when loading the key.

Example:

```
mail {
    ssl_password_file /etc/keys/global.pass;
    ...

    server {
        server_name mail1.example.com;
        ssl_certificate_key /etc/keys/first.key;
    }

    server {
        server_name mail2.example.com;

        # named pipe can also be used instead of a file
        ssl_password_file /etc/keys/fifo;
        ssl_certificate_key /etc/keys/second.key;
    }
}
```

ssl_prefer_server_ciphers

Syntax	`ssl_prefer_server_ciphers on` | `off`;
Default	`ssl_prefer_server_ciphers off;`
Context	mail, server

Specifies that server ciphers should be preferred over client ciphers when the SSLv3 and TLS protocols are used.

ssl_protocols

Syntax	`ssl_protocols` [SSLv2] [SSLv3] [TLSv1] [TLSv1.1] [TLSv1.2] [TLSv1.3];
Default	`ssl_protocols TLSv1.2 TLSv1.3;`
Context	mail, server

Changed in version 1.2.0: `TLSv1.3` parameter added to default set.

Enables the specified protocols.

Important

The TLSv1.1 and TLSv1.2 parameters work only when OpenSSL 1.0.1 or higher is used.

The TLSv1.3 parameters works only when OpenSSL 1.1.1 or higher is used.

ssl_session_cache

Syntax	`ssl_session_cache off` | `none` | [builtin[:*size*]] [shared:*name*:*size*];
Default	`ssl_session_cache none;`
Context	mail, server

Sets the types and sizes of caches that store session parameters. A cache can be of any of the following types:

`off`	the use of a session cache is strictly prohibited: Angie explicitly tells a client that sessions may not be reused.
`none`	the use of a session cache is gently disallowed: Angie tells a client that sessions may be reused, but does not actually store session parameters in the cache.
`builtin`	a cache built in OpenSSL; used by one worker process only. The cache size is specified in sessions. If size is not given, it is equal to 20480 sessions. Use of the built-in cache can cause memory fragmentation.
`shared`	a cache shared between all worker processes. The cache size is specified in bytes; one megabyte can store about 4000 sessions. Each shared cache should have an arbitrary name. A cache with the same name can be used in several servers. It is also used to automatically generate, store, and periodically rotate TLS session ticket keys unless configured explicitly using the *ssl_session_ticket_key* directive.

Both cache types can be used simultaneously, for example:

```
ssl_session_cache builtin:1000 shared:SSL:10m;
```

but using only shared cache without the built-in cache should be more efficient.

ssl_session_ticket_key

Syntax	`ssl_session_ticket_key` *file*;
Default	—
Context	mail, server

Sets a file with the secret key used to encrypt and decrypt TLS session tickets. The directive is necessary if the same key has to be shared between multiple servers. By default, a randomly generated key is used.

If several keys are specified, only the first key is used to encrypt TLS session tickets. This allows configuring key rotation, for example:

```
ssl_session_ticket_key current.key;
ssl_session_ticket_key previous.key;
```

The file must contain 80 or 48 bytes of random data and can be created using the following command:

```
openssl rand 80 > ticket.key
```

Depending on the file size either AES256 (for 80-byte keys) or AES128 (for 48-byte keys) is used for encryption.

ssl_session_tickets

Syntax	`ssl_session_tickets on` | `off`;
Default	`ssl_session_tickets on;`
Context	mail, server

Enables or disables session resumption through TLS session tickets.

ssl_session_timeout

Syntax	`ssl_session_timeout` *time*;
Default	`ssl_session_timeout 5m;`
Context	mail, server

Specifies a time during which a client may reuse the session parameters.

ssl_trusted_certificate

Syntax	`ssl_trusted_certificate` *file*;
Default	—
Context	mail, server

Specifies a file with trusted CA certificates in the PEM format used to *verify* client certificates.

In contrast to the certificate set by *ssl_client_certificate*, the list of these certificates will not be sent to clients.

ssl_verify_client

Syntax	`ssl_verify_client on` | `off` | `optional` | `optional_no_ca`;
Default	`ssl_verify_client off;`
Context	mail, server

Enables verification of client certificates. The verification result is passed in the "Auth-SSL-Verify" header of the *authentication* request. If an error occurs during client certificate verification or a client does not provide the required certificate, the connection is closed.

`optional`	requests the client certificate and verifies it if the certificate is present
`optional_no_ca`	requests the client certificate but does not require it to be signed by a trusted CA certificate. This is intended for the use in cases when a service that is external to Angie performs the actual certificate verification. The contents of the certificate is accessible through requests *sent* to the authentication server.

ssl_verify_depth

Syntax	`ssl_verify_depth` *number*;
Default	`ssl_verify_depth 1;`
Context	mail, server

Sets the verification depth in the client certificates chain.

starttls

Syntax	`starttls on` | `off` | `only`;
Default	`starttls off;`
Context	mail, server

`on`	allow usage of the STLS command for the POP3 and the STARTTLS command for the IMAP and SMTP;
`off`	deny usage of the STLS and STARTTLS commands;
`only`	require preliminary TLS transition.

The core mail module implements basic functionality for a mail proxy server: this includes support for SMTP, IMAP, and POP3 protocols, configuring server blocks, mail request routing, user authentication, and SSL/TLS support for securing mail connections.

The other modules in this section extend this functionality, allowing you to flexibly configure and optimize the mail server for various scenarios and requirements.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-mail` build option. In packages and images from our repos, the module is included in the build.

Configuration Example

```
worker_processes auto;

error_log /var/log/angie/error.log info;

events {
    worker_connections  1024;
}

mail {
    server_name       mail.example.com;
    auth_http         localhost:9000/cgi-bin/auth.cgi;

    imap_capabilities IMAP4rev1 UIDPLUS IDLE LITERAL+ QUOTA;

    pop3_auth         plain apop cram-md5;
    pop3_capabilities LAST TOP USER PIPELINING UIDL;

    smtp_auth         login plain cram-md5;
    smtp_capabilities "SIZE 10485760" ENHANCEDSTATUSCODES 8BITMIME DSN;
    xclient           off;

    server {
        listen   25;
        protocol smtp;
    }
    server {
        listen   110;
        protocol pop3;
        proxy_pass_error_message on;
    }
    server {
        listen   143;
        protocol imap;
    }
    server {
        listen   587;
        protocol smtp;
    }
}
```

Directives

listen

Syntax	`listen` *address*[:*port*] [`ssl`] [`proxy_protocol`] [`backlog=`*number*] [`rcvbuf=`*size*] [`sndbuf=`*size*] [`bind`] [`ipv6only=on` | `off`] [`reuseport`] [`so_keepalive=`on|off|[*keepidle*]:[*keepintvl*]:[*keepcnt*]];
Default	—
Context	server

Sets the *address* and *port* for the socket on which the server will accept requests. It is possible to specify just the *port*, so Angie listens on all available IPv4 (and IPv6, if enabled) interfaces. The address can also be a hostname, for example:

```
listen 127.0.0.1:110;
listen *:110;
listen 110;      # same as *:110
listen localhost:110;
```

IPv6 addresses are specified in square brackets:

```
listen [::1]:110;
listen [::]:110;
```

UNIX domain sockets are specified with the `unix:` prefix:

```
listen unix:/var/run/angie.sock;
```

Important

Different servers must listen on different *address:port* pairs.

`ssl`	allows specifying that all connections accepted on this port should work in SSL mode.
`proxy_protocol`	allows specifying that all connections accepted on this port should use the PROXY protocol. Obtained information is passed to the *authentication server* and can be used to *change the client address*.

The `listen` directive can have several additional parameters specific to socket-related system calls.

`backlog=`*number*	sets the *backlog* parameter in the `listen()` call that limits the maximum length for the queue of pending connections. By default, `backlog` is set to `-1` on FreeBSD, DragonFly BSD, and macOS, and to 511 on other platforms.
`rcvbuf=`*size*	sets the receive buffer size (the `SO_RCVBUF` option) for the listening socket.
`sndbuf=`*size*	sets the send buffer size (the `SO_SNDBUF` option) for the listening socket.
`bind`	this parameter instructs to make a separate `bind()` call for a given *address:port* pair. The fact is that if there are several *listen* directives with the same *port* but different addresses, and one of the `listen` directives listens on all addresses for the given port (`*:port`), Angie will `bind()` only to **:port*. It should be noted that the `getsockname()` system call will be made in this case to determine the address that accepted the connection. If the `backlog`, `rcvbuf`, `sndbuf`, `ipv6only`, `reuseport`, or `so_keepalive` parameters are used then for a given *address:port* pair a separate `bind()` call will always be made.
`ipv6only=on` | `off`	this parameter determines (via the *IPV6_V6ONLY* socket option) whether an IPv6 socket listening on a wildcard address *[::]* will accept only IPv6 connections or both IPv6 and IPv4 connections. This parameter is turned on by default. It can only be set once on start.

`so_keepalive=on` | `off` | `[keepidle]:[keepintvl]:[keepcnt]` configures the "TCP keepalive" behavior for the listening socket.

`''`	if this parameter is omitted then the operating system's settings will be in effect for the socket
`on`	the *SO_KEEPALIVE* option is turned on for the socket
`off`	the *SO_KEEPALIVE* option is turned off for the socket

Some operating systems support setting of TCP keepalive parameters on a per-socket basis using the `TCP_KEEPIDLE`, `TCP_KEEPINTVL`, and `TCP_KEEPCNT` socket options. On such systems (currently, Linux 2.4+, NetBSD 5+, and FreeBSD 9.0-STABLE), they can be configured using the keepidle, keepintvl, and keepcnt parameters. One or two parameters may be omitted, in which case the system default setting for the corresponding socket option will be in effect.

For example,

```
so_keepalive=30m::10
```

will set the idle timeout (*TCP_KEEPIDLE*) to 30 minutes, leave the probe interval (*TCP_KEEPINTVL*) at its system default, and set the probes count (*TCP_KEEPCNT*) to 10 probes.

mail

Syntax	`mail { ... }`
Default	—
Context	main

Provides the configuration file context in which the mail server directives are specified.

max_commands

Added in version 1.7.0.

Syntax	`max_commands` *number*;
Default	`max_commands 1000;`
Context	mail, server

Sets the maximum number of commands issued during authentication to enhance protection against DoS attacks.

max_errors

Syntax	`max_errors` *number*;
Default	`max_errors 5;`
Context	mail, server

Sets the number of protocol errors after which the connection is closed.

protocol

Syntax	`protocol` *imap* | *pop3* | *smtp;*
Default	—
Context	server

Sets the protocol for a proxied server. Supported protocols are *IMAP*, *POP3*, and *SMTP*.

If the directive is not set, the protocol can be detected automatically based on the well-known port specified in the `listen` directive:

```
imap: 143, 993
pop3: 110, 995
smtp: 25, 587, 465
```

When building from source, unnecessary protocols can be disabled using the `--without-mail_imap_module`, `--without-mail_pop3_module`, and `--without-mail_smtp_module` build options.

resolver

Syntax	`resolver` *address* ... [`valid=`*time*] [`ipv4=on` | `off`] [`ipv6=on` | `off`] [`status_zone=`*zone*];
Default	`resolver off;`
Context	mail, server

Configures name servers used to find the client's hostname to pass it to the *authentication server*, and in the *XCLIENT* command when proxying SMTP. For example:

```
resolver 127.0.0.53 [::1]:5353;
```

The address can be specified as a domain name or IP address, with an optional port. If port is not specified, the port 53 is used. Name servers are queried in a round-robin fashion.

By default, Angie caches answers using the TTL value of a response. The optional valid parameter allows overriding it:

`valid`	*optional* valid parameter allows overriding cached entry validity

```
resolver 127.0.0.53 [::1]:5353 valid=30s;
```

By default, Angie will look up both IPv4 and IPv6 addresses while resolving.

`ipv4=off`	disables looking up of IPv4 addresses
`ipv6=off`	disables looking up of IPv6 addresses
`status_zone`	*optional* parameter, enables statistics collection for specified zone

Tip

To prevent DNS spoofing, it is recommended configuring DNS servers in a properly secured trusted local network.

Tip

When running in Docker, use its internal DNS server address such as `127.0.0.11`.

resolver_timeout

Syntax	`resolver_timeout` *time;*
Default	`resolver_timeout 30s;`
Context	mail, server

Sets a timeout for DNS operations, for example:

```
resolver_timeout 5s;
```

server

Syntax	`server { ... }`
Default	—
Context	mail

Sets the configuration for a server.

server_name

Syntax	`server_name` *name*;
Default	`server_name hostname`;
Context	mail, server

Sets the server name that is used:

- in the initial POP3/SMTP server greeting;
- in the salt during the SASL CRAM-MD5 authentication;
- in the EHLO command when connecting to the SMTP backend, if the passing of the *XCLIENT* command is enabled.

If the directive is not specified, the machine's *hostname* is used.

timeout

Syntax	`timeout` *time;*
Default	`timeout 60s;`
Context	mail, server

Sets the timeout that is used before proxying to the backend starts.

Google PerfTools

The module enables profiling of Angie worker processes using Google Performance Tools. The module is intended for Angie developers and allows them to analyze and optimize the performance of the server by providing detailed insights into memory usage, CPU usage, and other performance-related metrics.

When building from the source code, this module isn't built by default; it should be enabled with the `--with-google_perftools_module` build option.

Important

This module requires the gperftools library.

Configuration Example

```
google_perftools_profiles /var/log/angie/perftools;
```

Profiles will be stored as `/var/log/angie/perftools.<worker PID>` files.

Directives

google_perftools_profiles

Syntax	`google_perftools_profiles` *filename prefix*;
Default	—
Context	main

Sets the prefix of the filename where the profiling information for the worker process Angie will be stored. The worker process ID is appended at the end of the name after a dot, for example: `/var/log/angie/perftools.1234`.

WASM Module

WAMR

The module enables integration with WebAssembly Micro Runtime for executing WASM code, adding a number of runtime-specific directives to the *wasm_modules* context.

In our repositories, the module is built dynamically and is available as a separate package named `angie-module-wamr`.

Configuration Example

```
wasm_modules {

    wamr_heap_size 16k;
```

```
    wamr_stack_size 16k;

    load fft_transform.wasm id=fft;
}
```

Directives

wamr_heap_size

Syntax	`wamr_heap_size` *size*;
Default	`wamr_heap_size 8k;`
Context	wasm_modules

Sets the heap *size* for an individual module instance.

wamr_global_heap_size

Syntax	`wamr_global_heap_size` *size*;
Default	`wamr_global_heap_size 1m;`
Context	wasm_modules

Sets the heap *size* for the entire WAMR runtime.

wamr_stack_size

Syntax	`wamr_stack_size` *size*;
Default	`wamr_stack_size 8k;`
Context	wasm_modules

Sets the stack *size* for an individual module instance.

Wasmtime

The module enables integration with the Wasmtime runtime for executing WASM code, adding a number of runtime-specific directives to the *wasm_modules* context.

In our repositories, the module is built dynamically and is available as a separate package named `angie-module-wasmtime`.

Configuration Example

```
wasm_modules {

    wasmtime_stack_size 8k;

    wasmtime_enable_wasi on;

    load fft_transform.wasm id=fft;
}
```

Directives

wasmtime_enable_wasi

Syntax	`wasmtime_enable_wasi on | off;`
Default	`wasmtime_enable_wasi on;`
Context	wasm_modules

Enables or disables the use of WebAssembly System Interface APIs that provide basic POSIX-like functionality to WASM modules running on Angie.

> **Note**
>
> Angie-specific APIs can be whitelisted using the *load* directive.

wasmtime_stack_size

Syntax	`wasmtime_stack_size` *size*;
Default	`wasmtime_stack_size 8k;`
Context	wasm_modules

Sets the max_wasm_stack value to a specific *size*, thus limiting the maximum amount of stack space available for executing WASM code.

The core module that implements basic WASM functionality in Angie: this includes support for loading alternative runtimes and WASM modules, as well as configuring their features and limits.

The other modules in this section extend this functionality, allowing you to flexibly configure and optimize the WASM features for various scenarios and requirements.

In our repositories, the module is built dynamically and is available as a separate package named `angie-module-wasm`.

Configuration Example

```
# These directives load the core functionality
load_module modules/ngx_wasm_module.so;
load_module modules/ngx_wasm_core_module.so;

load_module modules/ngx_wasmtime_module.so;

# Available here: https://git.angie.software/web-server/angie-wasm
load_module modules/ngx_http_wasm_host_module.so;

events {

}


wasm_modules {

    #use wasmtime;

    load ngx_http_handler.wasm id=handler;
    load ngx_http_vars.wasm id=vars type=reactor;
```

```
}


http {

    wasm_var vars "ngx:wasi/var-utils#sum-entry" $rvar $arg_a $arg_b $arg_c $arg_d;

    server {

        listen *:8080;

        location / {

            return 200 "sum('$arg_a','$arg_b','$arg_c','$arg_d')=$rvar\n";
        }

        location /wasm {

            client_max_body_size 20M;
            wasm_content handler "ngx:wasi/http-handler-entry#handle-request";
        }
    }
}
```

Directives

load

Syntax	`load` *file* `id=`*id* [`fs=`*host_path*`:`*guest_path*]... [`api=`*api*]... [`type=command` | `reactor`]
Default	—
Context	wasm_modules

Loads a module from a disk *file* and assigns it an *id* (required). During loading, the module is verified to ensure it can be instantiated.

The directive has the following parameters:

`fs`	Allows the guest to access a directory on the host. Can be specified multiple times for different directories.
`api`	Explicitly restricts the list of APIs available to the module by listing them. If the module attempts to use restricted APIs (i.e. not listed here), an "API not found" error will be returned. By default, the module has access to all APIs.
`type`	Controls the lifecycle of the loaded module. • In `command` mode, the machine runs once and its state is destroyed after execution. • In `reactor` mode, the machine effectively runs indefinitely, allowing multiple code executions. This requires careful memory management: if resources aren't cleaned up, memory leaks can occur.

wasm_modules

Syntax	`wasm_modules { ... };`
Default	—
Context	main

A top-level directive that provides the configuration file context in which the WASM directives should be specified. It can contain directives for loading WASM modules and configuring runtime-specific settings.

Core Module

Core	Responsible for managing service files, processes, and other Angie modules.

HTTP Modules

HTTP	Core functionality for processing HTTP requests and responses, managing the HTTP serve
Access	Access control based on IPs and CIDR ranges.
ACME	Automatic retrieval of SSL certificates using the ACME protocol.
Addition	Inserts a predefined snippet before or after the response body.
API	RESTful HTTP interface to obtain basic information about the web server and its statistic
Auth Basic	Basic HTTP authentication for access control based on the username and password.
Auth Request	Authorization using a subrequest to an external HTTP service.
AutoIndex	Automatic directory listing without an index file.
Browser (deprecated)	Browser identification based on the `User-Agent` header.
Charset	Configuration and conversion of response encoding.
DAV	File management on the server using the WebDAV protocol.
Empty GIF	Serves a one-pixel transparent GIF.
FastCGI	Proxying requests to a FastCGI server.
FLV	Pseudo-streaming of Flash Video (FLV) files.
Geo	Conversion of IP addresses into predefined variable values.
GeoIP	Retrieves IP address data using geolocation by MaxMind GeoIP databases.
gRPC	Proxying requests to a gRPC server.
GunZIP	Decompression of compressed GZip responses for modification or in cases where the client d
GZip	Compression of responses using the GZip method to save traffic.
GZip Static	Serves static files pre-compressed using the GZip method.
Headers	Modification of response header fields.
HTTP2	Handles requests using the HTTP/2 protocol.
HTTP3	Handles requests using the HTTP/3 protocol.
Image Filter	Image transformation.
Index	Configuration of index files that serve requests with a trailing slash (`/`).
JS	Handlers to extend functionality by implementing additional logic in njs, a subset of the Ja
Limit Conn	Limiting the number of concurrent requests (active connections) to protect against overloa
Limit Req	Limiting the request rate to protect against overload and password guessing.
Log	Configuration of request logs to track resource access for monitoring and analysis.
Map	Converts variables based on predefined key-value pairs.
Memcached	Retrieval of responses from a Memcached server.
Mirror	Mirroring requests to other servers.
MP4	Pseudo-streaming of MP4 files.
Perl	Handlers to extend functionality by implementing additional logic in the Perl language.
Prometheus	Server metrics in a Prometheus-compatible format for monitoring and statistics collection.
Proxy	Reverse proxying requests to other HTTP servers.
Random Index	Random selection of an index file for requests with a trailing slash (`/`).
RealIP	Client address and port identification when running behind another proxy server.
Referer	`Referer` header value validation.

Table 1 – continued from previous page

Rewrite	Conditional URI modification, redirects, variable setting, and configuration selection.
SCGI	Proxying requests to a SCGI server.
Secure Link	Secure link creation with the ability to restrict access time.
Slice	Splitting requests into multiple sub-requests for better caching of large responses.
Split Clients	Creating variables for A/B testing, canary releases, sharding, and other scenarios that requ
SSI	Processing SSI (Server Side Includes) commands in responses.
SSL	SSL/TLS configuration to handle HTTPS requests.
Stub Status (deprecated)	Global connection and request counters in a text-based format.
Sub	Search and replace snippets in server responses.
Upstream	Configures groups of proxied servers for load balancing.
Upstream Probe	Configures health probes for groups of proxied servers.
UserID	Issuing and processing unique client identifier cookies for session tracking and analytics.
uWSGI	Proxying requests to a uWSGI server.
XSLT	XML transformation using XSLT stylesheets.

Streaming Modules

Stream	Basic stream server functionality for balancing TCP and UDP protocols at the L4 level.
Access	Access control based on IPs and CIDR ranges.
Geo	Conversion of IP addresses into predefined variable values.
GeoIP	Retrieves IP address data using geolocation by MaxMind GeoIP databases.
JS	Handlers to extend functionality by implementing additional logic in njs, a subset of the JavaScript language.
Limit Conn	Limiting the number of concurrent requests (active connections) to protect against overload.
Log	Configuration of request logs to track resource access for monitoring and analysis.
Map	Converts variables based on predefined key-value pairs.
MQTT Preread	Reads the client identifier and username from an MQTT connection before making a load balancing decision.
Pass	Configures passing accepted connections directly to a configured listening socket.
Proxy	Configures proxying to other servers.
RDP Preread	Reads cookies from an RDP connection before making a load balancing decision.
RealIP	Client address and port identification when running behind another proxy server.
Return	Sends a specified value to the client upon connection without further proxying.
Set	Sets predefined variable values.
Split Clients	Creating variables for A/B testing, canary releases, sharding, and other scenarios that require a proportional group split.
SSL	Terminates SSL/TLS and DTLS protocols.
SSL Preread	Extracts information from the `ClientHello` message without terminating SSL/TLS and before making a load balancing decision.
Upstream	Configures groups of proxied servers for load balancing.
Upstream Probe	Configures health probes for groups of proxied servers.

Mail Modules

Mail	Basic mail proxy server functionality.
Auth HTTI	User authentication and server selection for subsequent proxying using HTTP requests to an external server.
IMAP	Support for the IMAP protocol.
POP3	Support for the POP3 protocol.
Proxy	Configures proxying to other servers.
RealIP	Client address and port identification when running behind another proxy server.
SMTP	Support for the SMTP protocol.
SSL	Support for the SSL/TLS and StartTLS protocols.

Google PerfTools Module

Google PerfToo	Integration with the Google Performance Tools library for application profiling and performance analysis.

WASM Modules

WASM	Basic WASM functionality to enable running WASM code in Angie.
WAMR	Integration with WebAssembly Micro Runtime.
Wasmtime	Integration with the Wasmtime runtime.

3.2.2 Built-in Variables

HTTP Modules	*Streaming Modules*
$acme_cert_key_<name>	
$acme_cert_<name>	
$ancient_browser	
$angie_version	*$angie_version*
$arg_<name>	
$args	
$binary_remote_addr	*$binary_remote_addr*
$body_bytes_sent	
	$bytes_received
$bytes_sent	*$bytes_sent*
$connection	*$connection*
$connection_requests	
$connection_time	
$connections_active	
$connections_reading	
$connections_writing	
$connections_waiting	
$content_length	
$content_type	
$cookie_<name>	
$date_local	
$date_gmt	
$document_root	
$document_uri	
$fastcgi_script_name	
$fastcgi_path_info	

continues on next page

Table 2 – continued from previous page

HTTP Modules	*Streaming Modules*
$gzip_ratio	
$host	
$hostname	*$hostname*
$http2	
$http3	
$http_<name>	
$https	
$invalid_referer	
$is_args	
$limit_conn_status	*$limit_conn_status*
$limit_rate	
$limit_req_status	
$memcached_key	
$modern_browser	
	$mqtt_preread_clientid
	$mqtt_preread_username
$msec	*$msec*
$msie	
$p8s_value	
$pid	*$pid*
$pipe	
$proxy_add_x_forwarded_for	
$proxy_host	
$proxy_port	
	$protocol
$proxy_protocol_addr	*$proxy_protocol_addr*
$proxy_protocol_port	*$proxy_protocol_port*
$proxy_protocol_server_addr	*$proxy_protocol_server_addr*
$proxy_protocol_server_port	*$proxy_protocol_server_port*
$proxy_protocol_tlv_<name>	*$proxy_protocol_tlv_<name>*
$query_string	
$quic_connection	
	$rdp_cookie, *$rdp_cookie_<name>*
$realip_remote_addr	*$realip_remote_addr*
$realip_remote_port	*$realip_remote_port*
$realpath_root	
$remote_addr	*$remote_addr*
$remote_port	*$remote_port*
$remote_user	
$request	
$request_body	
$request_body_file	
$request_completion	
$request_filename	
$request_id	
$request_length	
$request_method	
$request_time	
$request_uri	
$scheme	
$secure_link	
$secure_link_expires	
$sent_http_<name>	
$sent_trailer_<name>	
$server_addr	*$server_addr*

continues on next page

Table 2 – continued from previous page

HTTP Modules	*Streaming Modules*
$server_name	
$server_port	*$server_port*
$server_protocol	
	$session_time
$slice_range	
$ssl_alpn_protocol	*$ssl_alpn_protocol*
$ssl_cipher	*$ssl_cipher*
$ssl_ciphers	*$ssl_ciphers*
$ssl_client_escaped_cert	
	$ssl_client_cert
$ssl_client_fingerprint	*$ssl_client_fingerprint*
$ssl_client_i_dn	*$ssl_client_i_dn*
$ssl_client_i_dn_legacy	
$ssl_client_raw_cert	*$ssl_client_raw_cert*
$ssl_client_s_dn	*$ssl_client_s_dn*
$ssl_client_s_dn_legacy	
$ssl_client_serial	*$ssl_client_serial*
$ssl_client_v_end	*$ssl_client_v_end*
$ssl_client_v_remain	*$ssl_client_v_remain*
$ssl_client_v_start	*$ssl_client_v_start*
$ssl_client_verify	*$ssl_client_verify*
$ssl_curve	*$ssl_curve*
$ssl_curves	*$ssl_curves*
$ssl_early_data	*$ssl_early_data*
	$ssl_preread_alpn_protocols
	$ssl_preread_protocol
	$ssl_preread_server_name
$ssl_protocol	*$ssl_protocol*
$ssl_server_name	*$ssl_server_name*
$ssl_server_cert_type	*$ssl_server_cert_type*
$ssl_session_id	*$ssl_session_id*
$ssl_session_reused	*$ssl_session_reused*
$status	*$status*
$time_iso8601	*$time_iso8601*
$time_local	*$time_local*
$tcpinfo_rtt, *$tcpinfo_rttvar*, *$tcpinfo_snd_cwnd*, *$tcpinfo_rcv_space*	
$uid_got	
$uid_reset	
$uid_set	
$upstream_addr	*$upstream_addr*
$upstream_bytes_received	*$upstream_bytes_received*
$upstream_bytes_sent	*$upstream_bytes_sent*
$upstream_cache_status	
$upstream_connect_time	*$upstream_connect_time*
$upstream_cookie_<name>	
	$upstream_first_byte_time
$upstream_header_time	
$upstream_http_<name>	
$upstream_probe (PRO)	*$upstream_probe (PRO)*
$upstream_probe_body (PRO)	
	$upstream_probe_response (PRO)
$upstream_response_length	
$upstream_response_time	
	$upstream_session_time_<name>

continues on next page

Table 2 – continued from previous page

HTTP Modules	*Streaming Modules*
$upstream_status	
$upstream_sticky_status	*$upstream_sticky_status*
$upstream_trailer_<name>	
$upstream_queue_time	
$uri	

You can also use a short link service at https://angie.ws/en/ to quickly reference individual directives and variables:

3.2.3 Quick Access to Angie Directives and Variables

You can quickly access documentation for all Angie directives and variables without searching the site via our short link service at https://angie.ws/en/. It enables shortcuts to frequently used directives, variables and topics.

HTTP and Core Directives

Directives under *core settings* and *HTTP modules* use the `/h/` prefix (short for `http`).

Some examples:

- *listen*: https://angie.ws/en/h/listen
- *server*: https://angie.ws/en/h/server
- *worker_connections*: https://angie.ws/en/h/worker_connections

And so on.

> **Note**
>
> The *server* directive from the *Upstream* module is found at https://angie.ws/en/hu/server.

HTTP Upstream Directives

Directives in the *Upstream* module use the `/hu/` prefix (short for `http upstream`). Examples:

- *keepalive_requests*: https://angie.ws/en/hu/keepalive_requests
- *keepalive_time*: https://angie.ws/en/hu/keepalive_time
- *keepalive_timeout*: https://angie.ws/en/hu/keepalive_timeout

And so on.

Stream Directives

Directives in *Stream* modules use the `/s/` prefix (short for `stream`):

- *listen*: https://angie.ws/en/s/listen
- *map*: https://angie.ws/en/s/map
- *server*: https://angie.ws/en/s/server

And so on.

> **Note**
>
> The *server* directive from the *Upstream* module is found at https://angie.ws/en/su/server.

Stream Upstream Directives

Directives in the *Upstream* module use the `/su/` prefix (short for `stream upstream`):

- *upstream*: https://angie.ws/en/su/upstream
- *server*: https://angie.ws/en/su/server
- *state (PRO)*: https://angie.ws/en/su/state

And so on.

Variables

Variables follow the same prefix scheme.

HTTP variables (`/h/` prefix):

- *$angie_version*: https://angie.ws/en/h/\protect\T2A\textdollarangie_version
- *$upstream_status*: https://angie.ws/en/h/\protect\T2A\textdollarupstream_status

Stream variables (`/s/` prefix):

- *$angie_version*: https://angie.ws/en/s/\protect\T2A\textdollarangie_version
- *$upstream_session_time_<name>*: https://angie.ws/en/s/\protect\T2A\textdollarupstream_session_time

For placeholder variables like `$http_<HEADER>` or `$cookie_<NAME>`, use the prefix up to the underscore: https://angie.ws/en/h/\protect\T2A\textdollarhttp.

Additional Topics

Short links are also available for a number of other topics:

- Certificate Chaining
- Combined Locations
- Compact Server
- Configuration
- Configuration Hashes
- Configuration File Reloading
- Control Configuration Changes
- Control Signals
- Cyclic Memory Buffer
- Debug Logging
- Dummy Server
- HTTPS Configuration
- HTTPS Optimization
- HTTPS with Separate IPs
- HTTP Sessions
- Inheritance
- Load Balancing
- Location Redirect
- Log Rotation
- Method Usage

- Named Location
- Paths
- Picking a Location
- Proxy Pass URI
- Proxying
- Request Processing
- Runtime CLI Options
- Service Upgrade
- SNI (Server Name Indication)
- Source Build Features
- SSL Error Codes
- Stream Sessions
- Syntax
- Syslog Logging
- Virtual Server Selection
- WebSocket Proxy

3.3 How-tos

Instructions for specific aspects of configuring Angie are provided here.

3.3.1 How to migrate from nginx to Angie

If you're switching from nginx to Angie, congratulations! We have a guide for you.

Keep in mind, though, that it's tailored for a basic swap-out scenario that relies on a packaged version of Angie. If you're dealing with containers, virtual machines, custom paths, or modules, you'll need to tweak things accordingly.

Install Angie

We recommend using the official packages from our repos; see the instructions for your distribution to install Angie, but stop short of actually starting it. Instead, run `sudo angie -V` to make sure everything's in order:

```
$ sudo angie -V

  Angie version: Angie/1.9.1
  nginx version: nginx/1.27.4
  built by gcc 11.4.0
  configure arguments: --prefix=/etc/angie --conf-path=/etc/angie/angie.conf ...
```

As this output suggests, the *configuration* is located under `/etc/angie/` when Angie is installed from a package.

Update Angie configuration

Angie usually requires minimal changes to an existing nginx configuration.

1. Copy the entire nginx configuration to `/etc/angie/`:

```
$ sudo rsync -a --no-links /etc/nginx/ /etc/angie/
```

We assume that nginx stores its configuration under `/etc/nginx/`; adjust the steps if your path is different.

2. Rename the main configuration file as Angie expects it to be:

```
$ sudo mv /etc/angie/nginx.conf /etc/angie/angie.conf
```

3. Update the paths in the entire Angie configuration, starting with the main configuration file. The details vary depending on how nginx was installed, but at least you need to update the following parts.

   Any *include* paths that still point under `/etc/nginx/`:

```
# include /etc/nginx/conf.d/*.conf;
# include /etc/nginx/default.d/*.conf;
# include /etc/nginx/http.d/*.conf;
# include /etc/nginx/stream.d/*.conf;
include /etc/angie/conf.d/*.conf;
include /etc/angie/default.d/*.conf;
include /etc/angie/http.d/*.conf;
include /etc/angie/stream.d/*.conf;

# include /etc/nginx/sites-enabled/*;
include /etc/angie/sites-enabled/*;

# include /etc/nginx/modules-enabled/*;
include /etc/angie/modules-enabled/*;

# include /etc/nginx/mime.types;
include /etc/angie/mime.types;
```

   The *PID* file, which is crucial for Angie process management:

```
# pid /var/run/nginx.pid;
# -- or --
# pid /run/nginx.pid;
pid /run/angie.pid;
```

   Lastly, *access log* and *error log*:

```
# access_log /var/log/nginx/access.log;
access_log /var/log/angie/access.log;

# error_log /var/log/nginx/error.log;
error_log /var/log/angie/error.log;
```

Virtual hosts

If a directory such as `sites-enabled/` is used to include virtual hosts, update it:

```
# include /etc/nginx/sites-enabled/*;
include /etc/angie/sites-enabled/*;
```

Next, recreate the symlinks under `/etc/angie/sites-enabled/` to make it work.

List the original virtual host files, for example:

```
$ ls -l /etc/nginx/sites-enabled/

  default -> /etc/nginx/sites-available/default
```

Note their actual location; here, it's `/etc/nginx/sites-available/`.

If you didn't copy them under `/etc/angie/` previously, copy them now:

```
$ sudo rsync -a /etc/nginx/sites-available/ /etc/angie/sites-available/
```

Finally, recreate each symlink:

```
$ sudo ln -s /etc/angie/sites-available/default \
             /etc/angie/sites-enabled/default
```

Dynamic modules

Find and install Angie's counterparts to all dynamic modules referenced in nginx configuration, for example:

```
$ sudo nginx -T | grep load_module

  load_module modules/ngx_http_geoip2_module.so;
  load_module modules/ngx_stream_geoip2_module.so;
  ...
```

This means you need to install the `angie-module-geoip2` package, and so on.

There are two popular ways to include dynamic module configuration:

`/usr/share/nginx/modules/`

If the dynamic modules are included via `/usr/share/nginx/modules/`, update the path:

```
# Load dynamic modules. See /usr/share/doc/nginx/README.dynamic.
# include /usr/share/nginx/modules/*.conf;

include /usr/share/angie/modules/*.conf;
```

Next, copy the module configuration files:

```
$ sudo rsync -a /usr/share/nginx/modules/ /usr/share/angie/modules/
```

Lastly, amend the *load_module* path in each file:

```
# load_module "/usr/lib64/nginx/modules/ngx_http_geoip2_module.so";
load_module "/usr/lib64/angie/modules/ngx_http_geoip2_module.so";
```

`/etc/nginx/modules-enabled/`

If the `/etc/nginx/modules-enabled/` directory is used to include dynamic modules, update the path:

```
# include /etc/nginx/modules-enabled/*.conf;
include /etc/angie/modules-enabled/*.conf;
```

Next, recreate the symlinks under `/etc/angie/modules-enabled/` to make it work.

List the original module configuration files, for example:

```
$ ls -l /etc/nginx/modules-enabled/

  mod-http-geoip2.conf -> /usr/share/nginx/modules-available/mod-http-geoip2.conf
```

Note their actual location; here, it's `/usr/share/nginx/modules-available/`.

Copy them under `/usr/share/angie/`:

```
$ sudo rsync -a /usr/share/nginx/modules-available/ /usr/share/angie/modules-
↪available/
```

Finally, recreate each symlink:

```
$ sudo ln -s /usr/share/angie/modules-available/mod-http-geoip2.conf \
             /etc/angie/modules-enabled/mod-http-geoip2.conf
```

Root directory (optional)

If the *root* directive references a directory such as `/usr/share/nginx/html/`, you can change it to point to Angie.

Copy the directory and update the `root` setting in Angie configuration:

```
$ sudo rsync -a /usr/share/nginx/html/ /usr/share/angie/html/
```

```
# root /usr/share/nginx/html;
root /usr/share/angie/html;
```

User and group (optional)

While it's fine to leave the *user* directive as is, you can use separate accounts with Angie for extra flexibility.

Update the `user` settings in Angie configuration:

```
# user www-data www-data;
user angie angie;
```

Next, change the owner of *all* configuration files, including the ones under `/usr/share/angie/`, for example:

```
$ sudo chown -R angie:angie /etc/angie/
$ sudo chown -R angie:angie /usr/share/angie/
```

If Angie configuration defines the *root* directive, change the owner of the respective directories too, for example:

```
$ sudo chown -R angie:angie /var/www/html/
```

Wrapping up

To make sure you didn't miss anything, find and fix the remaining occurrences of `nginx` in Angie configuration:

```
$ grep -rn --include='*.conf' 'nginx' /etc/angie/
```

Test and switch

Now that you've updated Angie configuration, the next step is to verify its syntax to make sure Angie can actually work with it, and then perform the switchover. Check that Angie can run the updated configuration:

```
$ sudo angie -t
```

The command parses the configuration and reports issues that block Angie's start; fix any problems and re-run it.

Stop nginx, start Angie

To minimize downtime, start Angie immediately after stopping nginx:

```
$ sudo systemctl stop nginx && sudo systemctl start angie
```

Optionally, enable the Angie service to start it after reboot:

```
$ sudo systemctl enable angie
```

The migration is done! That's it. You're amazing.

Disable nginx

Last, when you're sure that Angie is running smoothly, you'll need to disable or uninstall nginx to avoid conflicts.

The least you can do is disable the service:

```
$ sudo systemctl disable nginx
```

Configure Angie extras

It's safe to assume that you're migrating for a reason, so why not go a step further and configure some of the additional features that Angie and Angie PRO offer on top of nginx?

3.3.2 ACME Configuration

The *ACME* module in Angie enables automatic certificate acquisition using the ACME protocol. The ACME protocol supports various domain verification methods (also called "validation"); this module implements *HTTP validation*, *DNS validation*, and *hook-based validation* through a custom external service.

Configuration Steps

General steps to enable certificate requests in the configuration:

- **Configure an ACME client** in the `http` block using the *acme_client* directive, which specifies a unique client name and other parameters. Multiple ACME clients can be configured.
- **Specify the domains for which certificates are requested**: A single certificate will be issued for all domain names listed in *server_name* directives within all `server` blocks that use *acme*, pointing to the same ACME client.
- **Set up request handling and ACME callbacks**: This is required to verify domain ownership. The setup depends on the chosen domain validation method:

Method	Requirements	Certificates
HTTP Validation	Requires responding to a specific request (callback) from the ACME server. Port 80 must be open on the Angie server.	Does not support wildcard certificates. Allows issuing multi-domain certificates.
DNS Validation	Requires handling specific DNS queries (callbacks) from the ACME server. Requires the ability to modify DNS records. The Angie server must have a port open as specified by the *acme_dns_port* directive (default is 53).	Supports issuing both multi-domain and wildcard certificates.
Hook-Based Validation	Requires implementing an external service that Angie communicates with. Requires an external DNS or HTTP server configured by the external service. The requirements for the external DNS or HTTP server match the respective points above.	Supports issuing both multi-domain and wildcard certificates.

- **Configure SSL using the obtained certificate and key**: Certificates and keys are made available as *embedded variables* that can be used in *configuration* to populate *ssl_certificate* and *ssl_certificate_key*.

  For SSL setup instructions, refer to *SSL Configuration*.

> **Tip**
>
> The certificate acquisition and renewal process depends on many services and may take some time. Be patient, and if you encounter problems or have doubts, check the *debug log*.

Implementation Details

Client keys and certificates are stored in PEM encoding within subdirectories of the directory specified by the `--http-acme-client-path` build option:

```
$ ls /var/lib/angie/acme/example/

  account.key  certificate.pem  private.key
```

The ACME client requires an account on the CA server, managed using a private key (`account.key`). If no key exists, it is generated at startup. The client registers the account with the CA server using this key.

> **Note**
>
> If you already have an account key, place it in the client's subdirectory before starting to reuse the account. Alternatively, specify the key using the `account_key` parameter in *acme_client*.

The client also uses a separate key (`private.key`) for Certificate Signing Requests (CSRs). This key is

automatically created if needed. At startup, the client requests a certificate by signing and sending a CSR for all domains under its management to the CA server.

The CA server verifies domain ownership using *HTTP* or *DNS validation* and issues a certificate, which is saved locally (`certificate.pem`).

As mentioned earlier, a single certificate covers all domains managed by the same ACME client, potentially resulting in a multi-domain certificate. The list of all covered names can be found in the *Subject Alternative Name* (SAN) field of the certificate. To check this in the terminal:

```
$ openssl x509 -in certificate.pem -noout -text | grep -A5 "Subject Alternative Name"
```

When a certificate is about to expire or domain changes occur, the client submits a new CSR, and the CA server re-verifies ownership before issuing a new certificate.

In the *configuration*, the obtained certificate and its corresponding key are available through the prefix variables *$acme_cert_<name>* and *$acme_cert_key_<name>*. Their values are the contents of the respective files, which should be used with the directives *ssl_certificate* and *ssl_certificate_key*:

```
server {

    listen 443 ssl;

    server_name example.com www.example.com;
    acme example;

    ssl_certificate $acme_cert_example;
    ssl_certificate_key $acme_cert_key_example;
}
```

HTTP Validation

Validation is handled automatically. The process involves the ACME server, upon receiving a request, retrieving a special file token via HTTP from the address `/.well-known/acme-challenge/<TOKEN>`. The ACME module intercepts and processes such requests automatically. When the expected response with the correct content is received, the ACME server confirms domain ownership.

Configuration Example

In this example, the ACME client named `example` manages the certificates for `example.com` and `www.example.com` (note that wildcards aren't supported with HTTP validation):

```
http {

    resolver 127.0.0.53; # Required for the 'acme_client' directive

    acme_client example https://acme-v02.api.letsencrypt.org/directory;

    server {

        listen 80; # May reside in a different 'server' block
        # with a different domain list
        # or even without one

        listen 443 ssl;

        server_name example.com www.example.com;
        acme example;

        ssl_certificate $acme_cert_example;
```

```
        ssl_certificate_key $acme_cert_key_example;
    }
}
```

As noted earlier, port 80 must be open to handle HTTP callbacks from ACME. However, as shown in this example, the *listen* directive for this port can be placed in a separate *server* block. If no existing block with this directive is present, a new block limited to ACME callbacks can be created:

```
server {

    listen 80;
    return 444; # No response, connection closed
}
```

Why does this work? The module intercepts requests to `/.well-known/acme-challenge/<TOKEN>` after reading headers but before selecting a virtual server or processing *rewrite* and *location* directives. Such intercepted requests are processed if the value of `<TOKEN>` matches the expected value for the specific callback. No directory access is performed; the module handles the request entirely.

DNS Validation

Validation is handled automatically. When processing a certificate request, the ACME server performs a special DNS query to the `_acme-challenge.` subdomain of the domain being verified. Once the expected response is received, the ACME server confirms domain ownership.

The ACME module tracks and processes such requests automatically, provided that your DNS records are configured properly to designate the Angie server as the authoritative name server for the `_acme-challenge.` subdomain.

For example, to verify the domain `example.com` using an Angie server at IP address `93.184.215.14`, your domain's DNS configuration should include the following records:

```
_acme-challenge.example.com. 60    IN      NS       ns.example.com.
            ns.example.com. 60    IN       A       93.184.215.14
```

This configuration delegates DNS resolution for `_acme-challenge.example.com` to `ns.example.com`, ensuring the availability of `ns.example.com` via its IP address (`93.184.215.14`).

This method allows requesting wildcard certificates, e.g. a certificate that will include the entry `*.example.com` in the *Subject Alternative Name* (SAN) section. To explicitly request a certificate for a subdomain, such as `www.example.com`, you must separately verify that subdomain using the method described above.

Attention

The applicability of this scenario largely depends on the capabilities of your DNS provider; some providers do not allow such configurations.

Configuration Example

The configuration is generally similar to the example in the previous section. There is no need for HTTP-specific settings; instead, set `challenge=dns` in the *acme_client* directive.

In this example, the ACME client named `example` manages the certificates for `example.com` and `*.example.com`:

```
http {

    resolver 127.0.0.53;
```

```
    acme_client example https://acme-v02.api.letsencrypt.org/directory
        challenge=dns;

    server {

        server_name example.com *.example.com;
        acme example;

        ssl_certificate $acme_cert_example;
        ssl_certificate_key $acme_cert_key_example;
    }
}
```

Hook-Based Validation

Unlike the previous methods, this validation requires additional effort. The ACME server performs a standard *HTTP validation* or *DNS validation*, but instead of interacting directly with the Angie server, it communicates with an external service managed by the Angie server using hooks (*acme_hook*). This service configures a separate DNS or HTTP server where the ACME server sends its requests.

Once the ACME server receives the expected response from the configured DNS or HTTP server, it confirms domain ownership.

Angie invokes the external service when certificate updates are needed via the ACME protocol. Requests and data are proxied to FastCGI, SCGI, or similar servers using directives such as *fastcgi_pass*, *scgi_pass*, etc.

The service must return a `2xx` status code, which can be sent via the `Status` header. Any other code is treated as an error, and certificate renewal is halted. Output from the service is ignored.

Configuration Example

In this example, the ACME client `example` is configured to use DNS-based validation, indicated by the `challenge=dns` parameter in the `acme_client` directive.

A `server` block applies to all subdomains of `example.com` (e.g., `*.example.com`) and uses the ACME client `example` to manage certificates, as specified by the `acme` directive.

A named `location` block is set up to handle external service calls for DNS validation. The `acme_hook` directive links the server to the ACME client `example`. Requests are proxied to a local FastCGI server running on port 9000 using the `fastcgi_pass` directive. The `fastcgi_param` directives pass data about the ACME client, hook, challenge type, domain, token, and key authorization to the external service.

```
acme_client example https://acme-v02.api.letsencrypt.org/directory
    challenge=dns;

server {

    listen 80;

    server_name *.example.com;

    acme example;

    ssl_certificate $acme_cert_example;
    ssl_certificate_key $acme_cert_key_example;

    location @acme_hook_location {
```

```
        acme_hook example;

        fastcgi_pass localhost:9000;

        fastcgi_param ACME_CLIENT $acme_hook_client;
        fastcgi_param ACME_HOOK $acme_hook_name;
        fastcgi_param ACME_CHALLENGE $acme_hook_challenge;
        fastcgi_param ACME_DOMAIN $acme_hook_domain;
        fastcgi_param ACME_TOKEN $acme_hook_token;
        fastcgi_param ACME_KEYAUTH $acme_hook_keyauth;

        include fastcgi.conf;
    }
}
```

The following Perl script demonstrates a simple external FastCGI service:

```
#!/usr/bin/perl

use strict; use warnings;

use FCGI;

my $socket = FCGI::OpenSocket(":9000", 5);
my $request = FCGI::Request(\*STDIN, \*STDOUT, \*STDERR, \%ENV, $socket);

while ($request->Accept() >= 0) {
    print "\r\n";

    my $client =    $ENV{ACME_CLIENT};
    my $hook =      $ENV{ACME_HOOK};
    my $challenge = $ENV{ACME_CHALLENGE};
    my $domain =    $ENV{ACME_DOMAIN};
    my $token =     $ENV{ACME_TOKEN};
    my $keyauth =   $ENV{ACME_KEYAUTH};

    if ($hook eq 'add') {

        DNS_set_TXT_record("_acme-challenge.$domain.", $keyauth);

    } elsif ($hook eq 'remove') {

        DNS_clear_TXT_record("_acme-challenge.$domain.");
    }
};

FCGI::CloseSocket($socket);
```

Here, `DNS_set_TXT_record()` and `DNS_clear_TXT_record()` are functions assumed to add and remove TXT records in the configuration of an external DNS server that the ACME server queries. These records must contain the data provided by the Angie server to allow successful validation, similar to the process described in *DNS Validation*. The details of implementing such functions are beyond the scope of this guide; for example, parameters can also be passed through the request URI:

```
# ...

location @acme_hook_location {
```

```
    acme_hook example uri=/acme_hook/$acme_hook_name?domain=$acme_hook_domain&key=
↪$acme_hook_keyauth;

    fastcgi_pass localhost:9000;

    fastcgi_param REQUEST_URI $request_uri;
    fastcgi_param ACME_CLIENT $acme_hook_client;
    fastcgi_param ACME_CHALLENGE $acme_hook_challenge;
    fastcgi_param ACME_TOKEN $acme_hook_token;

    include fastcgi.conf;
}
```

Another example using PHP-FPM:

```
location @acme_hook_location {

    acme_hook example;
    root /var/www/dns;
    fastcgi_pass unix:/run/php-fpm/php-dns.sock;
    fastcgi_index hook.php;
    fastcgi_param SCRIPT_FILENAME /var/www/dns/hook.php;
    include fastcgi_params;

    fastcgi_param ACME_CLIENT $acme_hook_client;
    fastcgi_param ACME_HOOK $acme_hook_name;
    fastcgi_param ACME_CHALLENGE $acme_hook_challenge;
    fastcgi_param ACME_DOMAIN $acme_hook_domain;
    fastcgi_param ACME_TOKEN $acme_hook_token;
    fastcgi_param ACME_KEYAUTH $acme_hook_keyauth;
}
```

```
[dns]
listen = /run/php-fpm/php-dns.sock
listen.mode = 0666
user = angie
group = angie
chdir = /var/www/dns
# ...
```

Parameters passed can be accessed in PHP via `$_SERVER['...']`.

Migrating from `certbot`

If you previously used certbot to obtain and renew SSL certificates from Let's Encrypt before *migrating from nginx to Angie*, follow these steps to transition to using the ACME module.

Suppose you initially configured certificates with the following command:

```
$ sudo certbot --nginx -d example.com -d www.example.com
```

The configuration automatically created by certbot is typically located in `/etc/nginx/sites-available/example.conf` and looks something like this:

```
server {

    listen 80;
    server_name example.com www.example.com;
    return 301 https://$host$request_uri;
```

```
}

server {

    listen 443 ssl;
    server_name example.com www.example.com;

    root /var/www/example;
    index index.html;

    ssl_certificate /etc/letsencrypt/live/example.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/example.com/privkey.pem;
    include /etc/letsencrypt/options-ssl-nginx.conf;
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem;

    location / {

        try_files $uri $uri/ =404;
    }
}
```

In the example above, the highlighted lines must be modified. Depending on your circumstances and preferences, configure *HTTP validation* or *DNS validation* using the ACME module.

The resulting *Angie configuration* might look like this:

```
http {

    resolver 127.0.0.53;

    acme_client example https://acme-v02.api.letsencrypt.org/directory;

    server {

        listen 80;
        server_name example.com www.example.com;
        return 301 https://$host$request_uri;
    }

    server {
        listen 443 ssl;
        server_name example.com www.example.com;

        root /var/www/example;
        index index.html;

        acme                 example;

        ssl_certificate      $acme_cert_example;
        ssl_certificate_key  $acme_cert_key_example;

        location / {
            try_files $uri $uri/ =404;
        }
    }
}
```

Remember to reload the configuration *after making changes*:

```
$ sudo kill -HUP $(cat /run/angie.pid)
```

Once the new configuration is verified, you can delete the `certbot` certificates and optionally disable or remove certbot entirely, if it is no longer used elsewhere:

```
$ sudo rm -rf /etc/letsencrypt

$ sudo systemctl stop certbot.timer
$ sudo systemctl disable certbot.timer
$ # -- or --
$ sudo rm /etc/cron.d/certbot

$ sudo apt remove certbot
$ # -- or --
$ sudo dnf remove certbot
```

3.3.3 SSL Configuration

To configure an HTTPS server, the *ssl* parameter must be enabled on listening sockets in the *server* block, and the locations of the server certificate and private key files should be specified:

```
server {
    listen              443 ssl;
    server_name         www.example.com;
    ssl_certificate     www.example.com.crt;
    ssl_certificate_key www.example.com.key;
    ssl_protocols       TLSv1.2 TLSv1.3;
    ssl_ciphers         HIGH:!aNULL:!MD5;
#...
}
```

The server certificate is a public entity. It is sent to every client that connects to the server. The private key is a secure entity and should be stored in a file with restricted access; however, it must be readable by Angie's master process. The private key may alternately be stored in the same file as the certificate.

```
ssl_certificate     www.example.com.cert;
ssl_certificate_key www.example.com.cert;
```

In which case the file access rights should also be restricted. Although the certificate and the key are stored in one file, only the certificate is sent to a client.

The directives *ssl_protocols* and *ssl_ciphers* can be used to limit connections to include only the strong versions and ciphers of SSL/TLS. By default, Angie uses:

```
ssl_protocols TLSv1.2 TLSv1.3;
ssl_ciphers HIGH:!aNULL:!MD5;
```

So configuring them explicitly is generally not needed.

HTTPS Server Optimization

SSL operations consume extra CPU resources. On multi-processor systems, several *worker processes* should be run, no less than the number of available CPU cores. The most CPU-intensive operation is the SSL handshake. There are two ways to minimize the number of these operations per client: the first is by enabling *keepalive* connections to send several requests via one connection, and the second is to reuse SSL session parameters to avoid SSL handshakes for parallel and subsequent connections. The sessions are stored in an SSL session cache shared between workers and configured by the *ssl_session_cache* directive. One megabyte of the cache contains about 4000 sessions. The default cache timeout is 5

minutes. It can be increased by using the *ssl_session_timeout* directive. Here is a sample configuration optimized for a multi-core system with a 10-megabyte shared session cache:

```
worker_processes auto;

http {
    ssl_session_cache   shared:SSL:10m;
    ssl_session_timeout 10m;

    server {
        listen              443 ssl;
        server_name         www.example.com;
        keepalive_timeout   70;

        ssl_certificate     www.example.com.crt;
        ssl_certificate_key www.example.com.key;
        ssl_protocols       TLSv1.2 TLSv1.3;
        ssl_ciphers         HIGH:!aNULL:!MD5;
    #...
```

Certificate Chains

Some browsers may complain about a certificate signed by a well-known certificate authority, while other browsers may accept the certificate without issues. This occurs because the issuing authority has signed the server certificate using an intermediate certificate that is not present in the certificate base of well-known trusted certificate authorities distributed with a particular browser. In this case, the authority provides a bundle of chained certificates which should be concatenated to the signed server certificate. The server certificate must appear before the chained certificates in the combined file:

```
$ cat www.example.com.crt bundle.crt > www.example.com.chained.crt
```

The resulting file should be used with the *ssl_certificate* directive:

```
server {
    listen              443 ssl;
    server_name         www.example.com;
    ssl_certificate     www.example.com.chained.crt;
    ssl_certificate_key www.example.com.key;
#...
}
```

If the server certificate and the bundle were concatenated in the wrong order, Angie fails to start and displays an error message:

> **SSL_CTX_use_PrivateKey_file(" ... /www.example.com.key") failed**
> (SSL: error:0B080074:x509 certificate routines: X509_check_private_key:key values mismatch)

Because Angie tried to use the private key with the bundle's first certificate instead of the server certificate.

Browsers usually store intermediate certificates that they receive, signed by trusted authorities, so browsers that are actually used may already have the required intermediate certificates and may not complain about a certificate being sent without a chained bundle. To ensure the server sends the complete certificate chain, the `openssl` command-line utility may be used.

```
$ openssl s_client -connect www.godaddy.com:443

Certificate chain
 0 s:/C=US/ST=Arizona/L=Scottsdale/1.3.6.1.4.1.311.60.2.1.3=US
```

```
     /1.3.6.1.4.1.311.60.2.1.2=AZ/O=GoDaddy.com, Inc
     /OU=MIS Department/CN=www.GoDaddy.com
     /serialNumber=0796928-7/2.5.4.15=V1.0, Clause 5.(b)
   i:/C=US/ST=Arizona/L=Scottsdale/O=GoDaddy.com, Inc.
     /OU=http://certificates.godaddy.com/repository
     /CN=Go Daddy Secure Certification Authority
     /serialNumber=07969287
 1 s:/C=US/ST=Arizona/L=Scottsdale/O=GoDaddy.com, Inc.
     /OU=http://certificates.godaddy.com/repository
     /CN=Go Daddy Secure Certification Authority
     /serialNumber=07969287
   i:/C=US/O=The Go Daddy Group, Inc.
     /OU=Go Daddy Class 2 Certification Authority
 2 s:/C=US/O=The Go Daddy Group, Inc.
     /OU=Go Daddy Class 2 Certification Authority
   i:/L=ValiCert Validation Network/O=ValiCert, Inc.
     /OU=ValiCert Class 2 Policy Validation Authority
     /CN=http://www.valicert.com//emailAddress=info@valicert.com
```

Tip

When testing configurations with *SNI*, it is important to specify the *-servername* option, as *openssl* does not use SNI by default.

In this example, the subject ("s") of the www.GoDaddy.com server certificate #0 is signed by an issuer ("i") which itself is the subject of the certificate #1, which is signed by an issuer which itself is the subject of the certificate #2, which is signed by the well-known issuer ValiCert, Inc. whose certificate is stored in the browsers' built-in certificate base.

If a certificate bundle has not been added, only the server certificate #0 will be shown.

A Single HTTP/HTTPS Server

It is possible to configure a single server that handles both HTTP and HTTPS requests:

```
server {
    listen              80;
    listen              443 ssl;
    server_name         www.example.com;
    ssl_certificate     www.example.com.crt;
    ssl_certificate_key www.example.com.key;
#...
}
```

Name-Based HTTPS Servers

A common issue arises when configuring two or more HTTPS servers listening on a single IP address:

```
server {
    listen          443 ssl;
    server_name     www.example.com;
    ssl_certificate www.example.com.crt;
#...
}

server {
    listen          443 ssl;
```

```
    server_name     www.example.org;
    ssl_certificate www.example.org.crt;
#...
}
```

With this configuration, a browser receives the default server's certificate, i.e. *www.example.com*, regardless of the requested server name. This is caused by SSL protocol behavior. The SSL connection is established before the browser sends an HTTP request, and Angie does not know the name of the requested server. Therefore, it may only offer the default server's certificate.

The oldest and most robust method to resolve the issue is to assign a separate IP address for every HTTPS server:

```
server {
    listen          192.168.1.1:443 ssl;
    server_name     www.example.com;
    ssl_certificate www.example.com.crt;
#...
}

server {
    listen          192.168.1.2:443 ssl;
    server_name     www.example.org;
    ssl_certificate www.example.org.crt;
#...
}
```

An SSL Certificate with Multiple Names

There are other ways that allow sharing a single IP address between several HTTPS servers. However, all of them have their drawbacks. One way is to use a certificate with several names in the `SubjectAltName` certificate field, for example, `www.example.com` and `www.example.org`. However, the `SubjectAltName` field length is limited.

Another way is to use a certificate with a wildcard name, for example, `*.example.org`. A wildcard certificate secures all subdomains of the specified domain, but only on one level. This certificate matches `www.example.org` but does not match `example.org` and `www.sub.example.org`. These two methods can also be combined. A certificate may contain exact and wildcard names in the `SubjectAltName` field, for example, `example.org` and `*.example.org`.

It is better to place a certificate file with several names and its private key file at the `http` level of configuration to inherit their single memory copy in all servers:

```
ssl_certificate     common.crt;
ssl_certificate_key common.key;

server {
    listen          443 ssl;
    server_name     www.example.com;
#...
}

server {
    listen          443 ssl;
    server_name     www.example.org;
#...
}
```

Server Name Indication

A more generic solution for running several HTTPS servers on a single IP address is TLS Server Name Indication extension (SNI, RFC 6066), which allows a browser to pass a requested server name during the SSL handshake, and therefore, the server will know which certificate it should use for the connection. SNI is currently supported by most modern browsers, though may not be used by some old or special clients.

> **Tip**
>
> Only domain names can be passed in SNI; however, some browsers may erroneously pass an IP address of the server as its name if a request includes a literal IP address. One should not rely on this.

If Angie was built with SNI support, then Ange will show this when run with the `-V` switch:

```
$ angie -V
...
TLS SNI support enabled
...
```

However, if the SNI-enabled Angie is linked dynamically to an OpenSSL library without SNI support, Angie displays a warning:

```
Angie was built with SNI support, however, now it is linked
dynamically to an OpenSSL library which has no tlsext support,
therefore SNI is not available
```

3.3.4 Console Light Web Monitoring Panel

Angie provides a wide range of possibilities to monitor its work; in addition to the *metrics* API and the *Prometheus* module, you can use a visual console that installs beside the server.

Console Light

Console Light is a lightweight, real-time activity monitoring interface that displays key server load and performance metrics. The console is based on the *API capabilities* of Angie; activity monitoring data is generated in real time. In addition, the console allows you to dynamically *modify* Angie configuration where the API itself provides this capability.

Example of a deployed and configured console: https://console.angie.software/

Version History

Version	Release Date	Changes
1.7.2	07.04.2025	Added "busy" option in filter controller on the "HTTP/TCP/UDP Upstreams" pages.
1.7.1	04.04.2025	Fixed incorrect values in the "HTTP/Location Zones" tables on the "HTTP Zones" page.
1.7.0	02.04.2025	• Display exact data volumes in bytes on mouse hover • New `busy` status for upstream peers in the statistics API, indicating that a peer has reached the limit configured by the `max_conns` parameter • Fixed documentation links
1.6.1	27.01.2025	• Fixed typos • Fixed a development-time project build issue
1.6.0	23.01.2025	• Internationalisation support with available locales: `en`, `ru`. • Sticky header feature added to the table component. • Support for data measurement units in pebibytes (PiB). • Fixed incorrect value counter in the *HTTP Upstreams* widget on the main page. • Default values are now correctly used on the *HTTP Upstreams* page in the response context.
1.5.0		Not publicly released.
1.4.0	08.08.2024	Added monitoring status display in the website favicon.
1.3.0	28.04.2024	Added the ability to set a server to the `draining` state in the upstream context.
1.2.1	26.12.2023	Added active health checks in the *Stream* context.
1.2.0	25.12.2023	Added server editing in the *Stream* context.

Installation and configuration

Console Light is published as `angie-console-light` (Angie) and `angie-pro-console-light` (Angie PRO) packages in our repositories and can be installed like any other package; alternatively, you can download the source code from our website or GitHub.

After installation, configure the console by adding the following *location* inside a *server* block in the *server configuration* (note the comments):

```
location /console/ {

    # Local access only
    allow 127.0.0.1;
    deny all;

    auto_redirect on;

    alias /usr/share/angie-console-light/html/;
    # FreeBSD only:
    # alias /usr/local/www/angie-console-light/html/;
    index index.html;

    location /console/api/ {
```

```
        api /status/;
    }

    # For editing features to work after authentication (PRO only)
    location /console/api/config/ {

        auth_basic           "Protected site";
        auth_basic_user_file conf/htpasswd;

        api /config/;
    }
}
```

Don't forget to apply the modified configuration:

```
$ sudo angie -t && sudo service angie reload
```

After this, the console will be available on the server specified by the `server` block, at the path specified for the `location`; in the example above, the path is set as `/console/`.

Authentication can be enabled for any API section similar to the example above, for instance:

```
location /console/server_zones/ {
    auth_basic           "Protected site";
    auth_basic_user_file conf/htpasswd;
}
```

You can also restrict access to any section of the configured console `location`, for example:

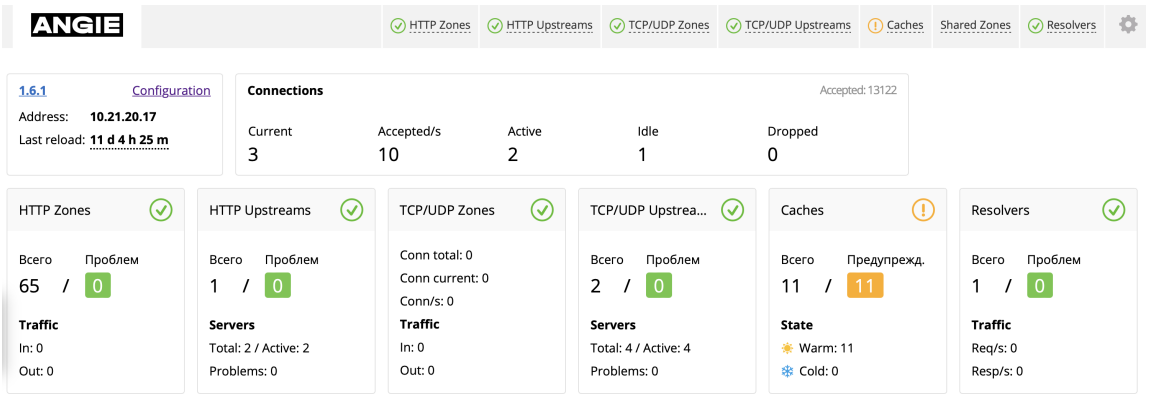
```
location /console/api/resolvers/ {
    deny all;
}
```

Interface

The console is a single screen with a set of tabs, each containing several widgets with monitoring data.

> **Hint**
>
> In the sections below, interface elements are described from left to right.

Angie Tab



This is the main tab where the key Angie monitoring indicators are displayed in summary form, based on data from several API sections.

About Widget

Displays the Angie version number with a link to the corresponding documentation, as well as the server address and the time of the last *configuration reload*.

Additionally, if the *api_config_files* directive is enabled, the *Configs* link opens a list of configuration files loaded on the server. Each file can then be viewed in a compact format with syntax highlighting.

Connections Widget

Displays basic server connection statistics, generated from the `/status/connections/` API section:

Current	Current number of connections
Accepted/s	Number of connections accepted per second
Active	Number of active connections
Idle	Number of idle connections
Dropped	Number of dropped connections

Also available:

Accepted	Total number of connections accepted since the last server reload

HTTP Zones Widget

Attention

Requires setting the *status_zone* directive in a `server` or `location` context.

Displays shared memory zone statistics for the `http` context, generated from the */status/http/server_zones/* API section:

Total	Total number of zones
Problems	Number of zones with any issues
Traffic	Total incoming and outgoing traffic volume

HTTP Upstreams Widget

Attention

Requires setting the *zone* directive in an *upstream* block in the `http` context.

Displays upstream statistics for the `http` context, generated from the */status/http/upstreams/* API section:

Total	Total number of upstreams
Problems	Number of upstreams with any issues
Servers	Server statistics broken down by state

TCP/UDP Zones Widget

Attention

Requires setting the following directives:

- `status_zone` in a *server* or *stream* context;
- `limit_conn` in a *server* or *stream* context;
- *limit_conn_zone* in the `stream` context.

Example:

```
stream {

    # ...
    limit_conn_zone $connection zone=limit-conn-stream:10m;

    server {

        # ...
        limit_conn limit-conn-stream 1;
        status_zone foo;
    }
}
```

Displays shared memory zone statistics for the `stream` context, generated from the */status/stream/server_zones/* API section:

Conn total	Total number of client connections
Conn current	Current number of client connections
Conn/s	Number of connections processed per second

TCP/UDP Upstreams Widget

Attention

Requires setting the *zone* directive in an *upstream* block in the `stream` context.

Displays upstream statistics for the `stream` context, generated from the */status/stream/upstreams/* API section:

Total	Total number of upstreams
Problems	Number of upstreams with any issues
Servers	Server statistics broken down by state

HTTP Zones Tab

Attention

Requires setting the *status_zone* directive in a `server` or `location` context.

Server Zones Section

Server Zones

Zone	Requests			Responses						Traffic				SSL			
	Current	Total	Req/s	1xx	2xx	3xx	4xx	5xx	Total	Sent/s	Rcvd/s	Sent	Rcvd	Handshaked	Reuses	Timedout	Failed
Brazil	1	1518366	3	0	1336935	2452	4880	174098	1518189	108 KiB	288 B	41.1 GiB	84.4 MiB	784	0	0	0
Russia	4	1382189	5	0	1373385	2504	5017	1279	1382010	61.4 KiB	228 B	42.1 GiB	73.7 MiB	836	0	0	0
India	4	1426856	4	0	1417678	2619	5266	1289	1426654	133 KiB	223 B	43.5 GiB	77.1 MiB	742	0	0	0
China	8	781161	2	0	729192	1375	3072	47514	780711	117 KiB	227 B	22.4 GiB	41.9 MiB	521	0	0	0
South Africa	5	1401366	6	0	1392465	2478	5113	1305	1401198	130 KiB	262 B	42.7 GiB	85.4 MiB	760	0	0	0
Argentina	4	1478295	7	0	1468815	2753	5449	1274	1478107	243 KiB	498 B	45.1 GiB	87.5 MiB	846	0	0	0
Egypt	2	1409034	3	0	1399957	2588	5155	1332	1408853	266 KiB	331 B	42.9 GiB	74.3 MiB	756	0	0	0
Ethiopia	2	1287945	5	0	1279661	2426	4676	1180	1287767	211 KiB	358 B	39.2 GiB	74.1 MiB	718	0	0	0
Iran	6	1414692	10	0	1405419	2564	5350	1353	1414478	391 KiB	754 B	43.1 GiB	73.0 MiB	830	0	0	0
Saudi Arabia	4	1469922	9	0	1460600	2605	5401	1312	1469717	367 KiB	617 B	44.8 GiB	87.0 MiB	818	0	0	0
UAE	1	1459720	2	0	1450440	2705	5258	1316	1459506	50.8 KiB	106 B	44.5 GiB	74.5 MiB	838	0	0	0

Summarizes shared memory zone monitoring statistics for the `server` context in `http`, generated from the */status/http/server_zones/* API section. The following data is presented for each zone:

Zone	Zone name **Hint** Click the arrow next to *Zone* to sort zones alphabetically or by configuration order.
Requests	Total number of requests and the number of requests per second
Responses	Number of responses broken down by status codes, as well as their total number
Traffic	Outgoing and incoming traffic rates, as well as total volumes of outgoing and incoming traffic
SSL	Aggregate counts of: successful SSL handshakes; SSL session reuses; SSL handshakes with expired timeout; unsuccessful SSL handshakes

Location Zones Section

Location Zones

Zone	Requests		Responses						Traffic			
	Total	Req/s	1xx	2xx	3xx	4xx	5xx	Total	Sent/s	Rcvd/s	Sent	Rcvd
Brasilia	308043	0	0	271399	509	955	35180	308005	38.9 KiB	57.0 B	8.33 GiB	17.0 MiB
Diadema	302843	0	0	266569	501	997	34776	302812	58.9 KiB	56.0 B	8.20 GiB	16.4 MiB
Porto Alegre	298320	2	0	262673	462	974	34211	298284	40.7 KiB	184 B	8.07 GiB	17.6 MiB
Salvador	303063	0	0	266622	487	954	35000	303026	0	0	8.18 GiB	16.7 MiB
Vitoria	306162	0	0	269732	493	1000	34937	306128	0	0	8.28 GiB	16.6 MiB
Lipetsk	280321	0	0	278579	502	978	262	280292	53.6 KiB	56.0 B	8.54 GiB	15.2 MiB
Moscow	274972	0	0	273224	479	1028	241	274935	15.6 KiB	55.0 B	8.38 GiB	14.7 MiB
Omsk	271726	0	0	270024	510	943	249	271688	0	0	8.29 GiB	14.0 MiB
Sevastopol	275881	0	0	274075	497	1047	262	275852	0	0	8.40 GiB	15.8 MiB
Ufa	279348	2	0	277546	516	1021	265	279306	132 KiB	157 B	8.52 GiB	14.1 MiB

Summarizes shared memory zone monitoring statistics for the `location` context in `http`, generated from the */status/http/location_zones/* API section. The following data is presented for each zone:

Zone	Zone name **Hint** Click the arrow next to *Zone* to sort zones alphabetically or by configuration order.
Requests	Total number of requests and the number of requests per second
Responses	Number of responses broken down by status codes, as well as their total number
Traffic	Outgoing and incoming traffic rates, as well as total volumes of outgoing and incoming traffic

Limit Conn Section

Displays statistics of `limit_conn` zones in the `http` context, generated from the */status/http/limit_conns/* API section. The following data is presented for each zone:

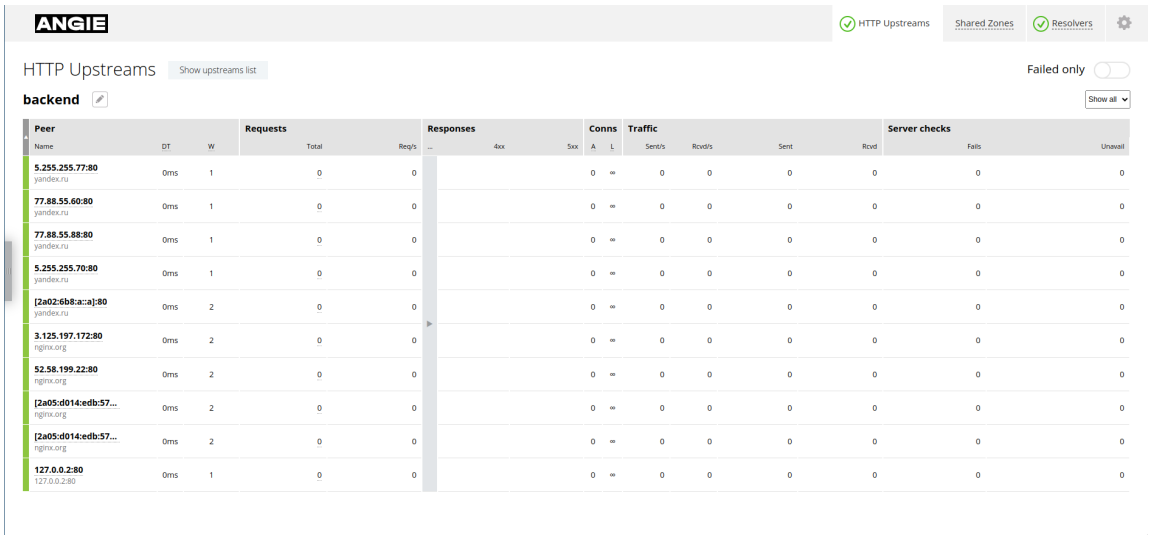
Zone	Zone name **Hint** Click the icon next to *Zone* to open or close the chart with the following indicators.
Passed	Total number of proxied connections
Rejected	Total number of rejected connections
Exhausted	Total number of connections dropped due to zone storage overflow
Skipped	Total number of connections passed with a zero or greater than 255 bytes key

Limit Req Section

Displays statistics of `limit_reqs` zones in the `http` context, generated from the */status/http/limit_reqs/* API section. The following data is presented for each zone:

Zone	Zone name **Hint** Click the icon next to *Zone* to open or close the chart with the following indicators.
Passed	Total number of proxied connections
Delayed	Total number of delayed connections
Rejected	Total number of rejected connections
Exhausted	Total number of connections dropped due to zone storage overflow
Skipped	Total number of connections passed with a zero or greater than 255 bytes key

HTTP Upstreams Tab



Attention

Requires setting the *zone* directive in an *upstream* block in the `http` context.

This tab summarizes upstream monitoring statistics for the `http` context, generated from the */status/http/upstreams/* API section.

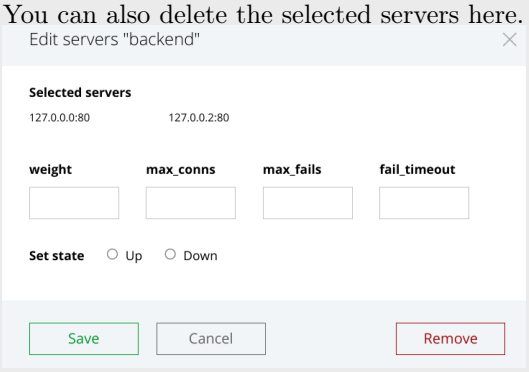
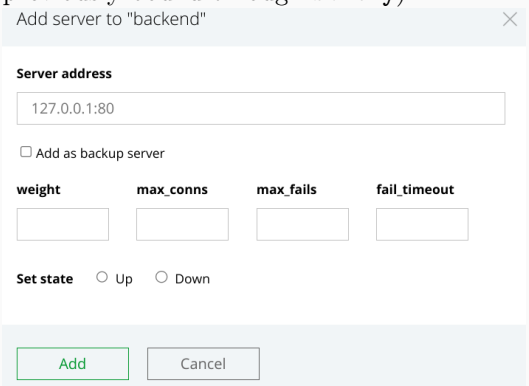
- The *Show upstreams list* button toggles a brief list of upstreams with the number of problematic upstreams and peers.
- The *Failed only* switch toggles the display mode for problematic upstreams statistics.
- The edit button toggles the `upstream editing` interface.
- The dropdown list on the right side of each upstream table allows you to filter servers in a specific state (*Up*, *Failed*, *Checking*, *Down*, *Busy*).

For each upstream, in addition to its name and shared memory zone utilization ratio, the following data is presented:

Server	Names, downtimes, and weights of upstream servers **Hint** Click the arrow next to *Server* to sort servers by their state or configuration order.
Requests	Total number and processing rate of requests
Responses	Number of responses broken down by status codes
Connections	Number of active connections and their maximum limit, if set
Traffic	Outgoing and incoming traffic rates, as well as total volumes of outgoing and incoming traffic
Server checks	Number of unsuccessful attempts to contact the server and the number of times the server was considered unavailable (the `health` object in the API)
Health monitors	Total number of server checks, number of unsuccessful checks, and the time of the last check

Editing upstreams

In Angie PRO, there is an edit button next to each upstream; when clicked, it displays two more buttons:

Edit selected	Edit selected servers within an upstream. Allows you to set the following parameters for all at once: *Weight*, maximum connection limit (*Max_ conns*), maximum failure limit that marks a server as unavailable (*Max_ fails*), time window for counting failures for the maximum failure limit (*Fail_ timeout*), state (`active` - enabled, `down` - disabled, or `draining` - only receives requests from sessions previously bound through `sticky`). You can also delete the selected servers here. 
Add server	Add a server to the upstream. Allows you to set the following parameters: address, backup server or not, *Weight*, maximum connection limit (*Max_ conns*), maximum failure limit that marks a server as unavailable (*Max_ fails*), failure counting time window (*Fail_ timeout*), state (`active` - enabled, `down` - disabled, or `draining` - only receives requests from sessions previously bound through `sticky`). 

TCP/UDP Zones Tab

Attention

Requires setting the following directives:

- `status_zone` in a *server* or *stream* context;
- `limit_conn` in a *server* or *stream* context;
- *limit_ conn_ zone* in the `stream` context.

Example:

```
stream {

    # ...
```

```
    limit_conn_zone $connection zone=limit-conn-stream:10m;

    server {

        # ...
        limit_conn limit-conn-stream 1;
        status_zone foo;
    }
}
```

TCP/UDP Zones Section

TCP/UDP Zones

Zone	Connections			Sessions				Traffic				SSL			
	Current	Total	Conn/s	2xx	4xx	5xx	Total	Sent/s	Rcvd/s	Sent	Rcvd	Handshakes	Handshakes Failed	Session Reuses	Verify Failures
sing_chorus	3	920	0	845	0	72	917	0	0	11.8 GiB	17.4 MiB	848	0	416	0

Summarizes shared memory zone monitoring statistics for the `server` context in `stream`, generated from the */status/stream/server_zones/* API section. The following data is presented for each zone:

Zone	Zone name
Connections	Current and total number of connections, as well as the number of connections per second
Sessions	Number of sessions broken down by status codes, as well as their total number
Traffic	Outgoing and incoming traffic rates, as well as total volumes of outgoing and incoming traffic
SSL	Aggregate counts of: successful SSL handshakes; unsuccessful SSL handshakes; SSL session reuses

Limit Conn Section

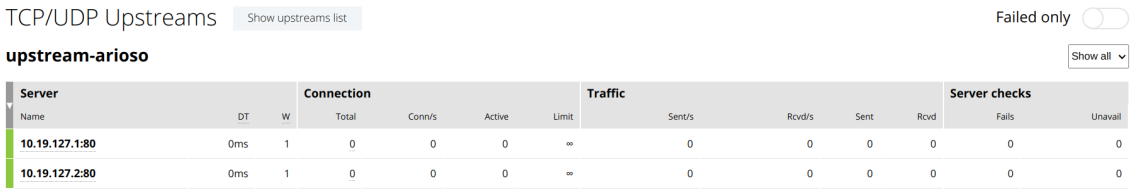
Limit Conn

	Zone	Passed	Rejected	Exhausted	Skipped
	limit-conn-stream	920	0	0	0

Displays statistics of `limit_conn` zones in the `stream` context, generated from the */status/stream/limit_conns/* API section. The following data is presented for each zone:

Zone	Zone name **Hint** Click the icon next to *Zone* to open or close the chart with the following indicators.
Passed	Total number of proxied connections
Rejected	Total number of rejected connections
Exhausted	Total number of connections dropped due to zone storage overflow
Skipped	Total number of connections passed with a zero or greater than 255 bytes key

TCP/UDP Upstreams Tab



Attention

Requires setting the *zone* directive in an *upstream* block in the `stream` context.

This tab summarizes upstream monitoring statistics for the `stream` context, generated from the */status/stream/upstreams/* API section.

- The *Show upstreams list* button toggles the display of a brief list of upstreams with the number of problematic upstreams and peers.
- The *Failed only* switch enables and disables the display mode for problematic upstreams statistics.
- The edit button opens the `upstream editing` widget.
- The dropdown list on the right side of each upstream table allows you to filter servers in a specific state (*Up*, *Failed*, *Checking*, *Down*, *Busy*).

For each upstream, the following data is presented:

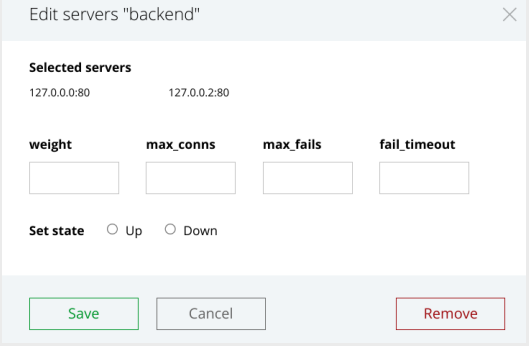
Server	Names, downtimes, and weights of upstream servers **Hint** Click the arrow next to *Server* to sort servers by their state or configuration order.
Connections	Number of active connections and their maximum limit, if set
Traffic	Outgoing and incoming traffic rates, as well as total volumes of outgoing and incoming traffic
Server checks	Number of unsuccessful attempts to contact the server and the number of times the server was considered unavailable (the `health` object in the API)

Editing upstreams

In Angie PRO, there is an edit button next to each upstream; when clicked, it displays two more buttons:

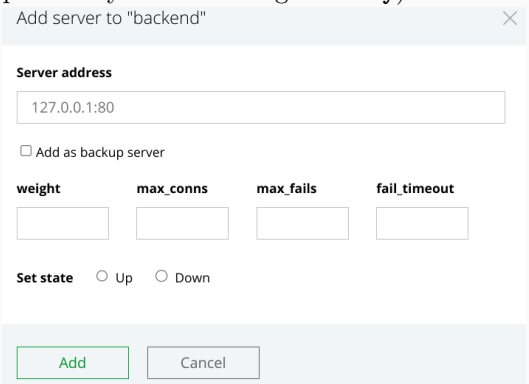
Edit selected

Edit selected servers within an upstream. Allows you to set the following parameters for all at once: *Weight*, maximum connection limit (*Max_conns*), maximum failure limit that marks a server as unavailable (*Max_fails*), time window for counting failures for the maximum failure limit (*Fail_timeout*), state (`active` - enabled, `down` - disabled, or `draining` - only receives requests from sessions previously bound through `sticky`).
You can also delete the selected servers here.

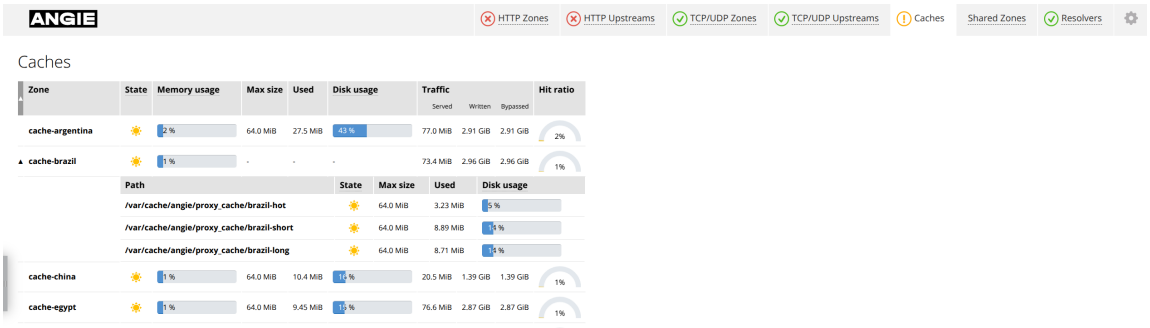


Add server

Add a server to the upstream. Allows you to set the following parameters: address, backup server or not, *Weight*, maximum connection limit (*Max_conns*), maximum failure limit that marks a server as unavailable (*Max_fails*), failure counting time window (*Fail_timeout*), state (`active` - enabled, `down` - disabled, or `draining` - only receives requests from sessions previously bound through `sticky`).



Caches Tab



> **Attention**
>
> Requires setting the *proxy_cache_path* directive in the `http` context.

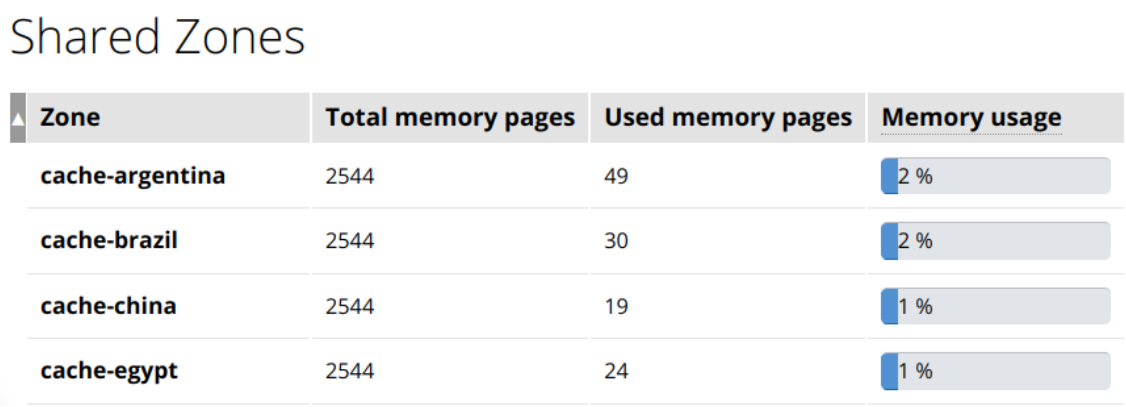
This tab summarizes monitoring statistics for `proxy_cache` zones in the `http` context, generated from the */status/http/caches/* API section. The following data is presented for each zone:

Zone	Zone name **Hint** Click the icon next to *Zone* to open or close the lists of *shards* for all zones that have them.
State	Cache state: cold (metadata being loaded into memory) or hot (metadata loaded)
Memory usage	Memory utilization ratio
Max size	Maximum memory size
Used	Used memory size
Disk usage	Disk utilization ratio
Traffic	Traffic served from cache, written to cache, and returned bypassing the cache
Hit ratio	Cache hit ratio (ratio of traffic served from cache to total volume)

If *sharding* is enabled for a zone, it is shown as a dropdown list that lists individual shards:

Path	Shard path on disk
State	Shard state: cold (metadata being loaded into memory) or hot (metadata loaded)
Max size	Maximum memory size
Used	Used memory size
Disk usage	Disk utilization ratio

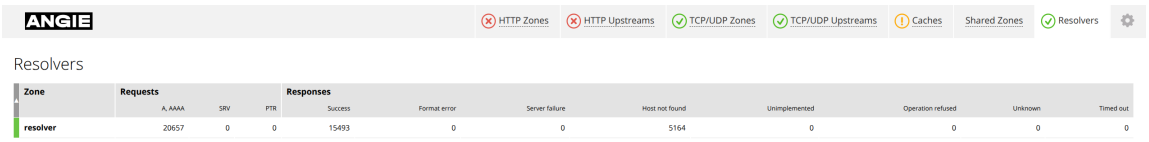
Shared Zones Tab



This tab summarizes monitoring statistics for **all** shared memory zones across all contexts. The following data is presented for each zone:

Zone	Zone name **Hint** Click the arrow next to *Zone* to sort zones by size or configuration order.
Total memory pages	Total number of memory pages
Used memory pages	Number of memory pages used
Memory usage	Memory utilization ratio for the zone

Resolvers Tab



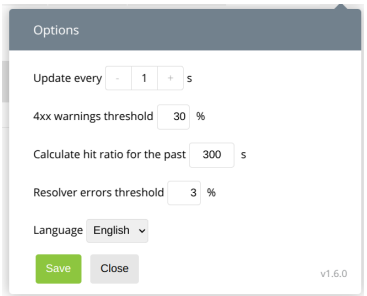
Attention

Requires setting the *resolver* directive in the `http` context.

This tab summarizes query statistics in DNS shared memory zones, generated from the */status/resolvers/* API section. The following data is presented for each zone:

Zone	Zone name **Hint** Click the arrow next to *Zone* to sort zones by state or configuration order.
Requests	Number of A and AAAA, SRV, PTR type requests
Responses	Number of responses broken down by corresponding codes (*Success*, *Format error*, *Server failure*, *Name error*, *Not implemented*, *Refused* and others)

Settings Widget



Allows you to configure general console parameters:

- Data refresh rate. Default value — 1 sec.
- Threshold ratio for `4xx` statuses. When the threshold is reached, "yellow" warnings appear in the corresponding sections related to server responses. Default value — 7%.

- Time window for calculating the cache hit ratio. Default value — 300 sec.
- Error threshold for the resolver. When the threshold is reached, the resolver will turn "red". Default value — 3%.
- Console interface language. Available options: English and Russian. By default, the console language is selected based on the locale set in the browser.

Console Control Panel

On all tabs, in the middle of the left side of the page, there is a slide-out panel with two buttons  .

The top button pauses and resumes data updates from the API, while the bottom button allows you to update the data manually when updates are paused.

3.3.5 Configuring the Prometheus dashboard

To configure the Prometheus dashboard for Angie in Grafana, follow these steps:

1. Using the the *Prometheus* module, add the following *include* directive in the `http` block of the *configuration file*:

   ```
   http {
       include prometheus_all.conf;

       # ...
   }
   ```

   And a respective *prometheus* directive inside the `location` within a separate `server` block that uses a designated IP address and port combination, for instance:

   ```
   server {

       listen 192.168.1.100:80;

       location =/p8s {
           prometheus all;
       }

       # ...

   }
   ```

   They enable the export of Angie metrics in Prometheus format at the endpoint specified by `location`.

2. Add the following configuration to Prometheus, specifying the IP address and port set earlier under `server`:

   ```
   scrape_configs:
     - job_name: "angie"
       scrape_interval: 15s
       metrics_path: "/p8s"
       static_configs:
         - targets: ["192.168.1.100:80"]
   ```

   It collects metrics every 15 seconds, utilizing the `/p8s` path configured at the previous step.

   Note

> Make sure the global `scrape_interval` value is not larger that the value here.

3. Import the Prometheus dashboard for Angie to Grafana.

CHAPTER 4

Troubleshooting

If you encounter a technical issue and can't find a solution in other sections, ask a question on the community forum or in the Telegram channel.

Technical support for clients:

- https://support.angie.software
- support@angie.software

4.1 Debug Logging

The debug log should be enabled before performing self-diagnostics or as recommended by support.

To do this, run Angie using the executable with debug logging enabled:

Linux

In the pre-built packages for Linux, the `angie-debug` file is built with debug logging enabled:

```
$ ls -l /usr/sbin/ | grep angie

   lrwxrwxrwx 1 root root      13 Sep 21 18:58 angie -> angie-nodebug
   -rwxr-xr-x 1 root root 1561224 Sep 21 18:58 angie-debug
   -rwxr-xr-x 1 root root 1426056 Sep 21 18:58 angie-nodebug
```

Configure running `angie-debug`:

```
$ sudo ln -fs angie-debug /usr/sbin/angie
$ sudo angie -t && sudo service angie upgrade
```

This will initiate a *live executable upgrade*.

To revert to the regular executable after debugging:

```
$ sudo ln -fs angie-nodebug /usr/sbin/angie
$ sudo angie -t && sudo service angie upgrade
```

FreeBSD

In the pre-built packages for FreeBSD, the `angie-debug` file is built with debug logging enabled:

```
$ ls -l /usr/local/sbin/ | grep angie

   lrwxrwxrwx 1 root root      13 Sep 21 18:58 angie -> angie-nodebug
   -rwxr-xr-x 1 root root 1561224 Sep 21 18:58 angie-debug
   -rwxr-xr-x 1 root root 1426056 Sep 21 18:58 angie-nodebug
```

Configure running `angie-debug`:

```
$ sudo ln -fs angie-debug /usr/local/sbin/angie
$ sudo angie -t && sudo service angie upgrade
```

This will initiate a *live executable upgrade*.

To revert to the regular executable after debugging:

```
$ sudo ln -fs angie-nodebug /usr/local/sbin/angie
$ sudo angie -t && sudo service angie upgrade
```

Building from Source

When building Angie from source, enable debugging before compilation:

```
$ ./configure --with-debug ...
```

After installation, `angie -V` allows verifying that debug logging is enabled:

```
$ angie -V

  ...
  configure arguments: --with-debug ...
```

Note

Using the executable with debug support enabled may slightly reduce performance; enabling the debug log can significantly reduce it and increase disk space usage.

To enable the debug log, set the `debug` level in the configuration using the *error_log* directive:

```
error_log /path/to/log debug;
```

And reload the configuration:

```
$ sudo angie -t && sudo service angie reload
```

Note

If you switch to the executable without debug support enabled but leave the `debug` level in the *error_log* directive, Angie will log entries at the `info` level.

Overriding *error_log* in the configuration without specifying the `debug` level disables the debug log. Here, overriding the log at the *server* level disables debug logging for an individual server:

```
error_log /path/to/log debug;

http {
   server {
```

```
    error_log /path/to/log;
   # ...
```

To avoid this, remove the line that overrides *error_log*, or set the `debug` level in it:

```
error_log /path/to/log debug;

http {
   server {
     error_log /path/to/log debug;
   #  ...
```

4.1.1 Logging Specific Addresses

You can enable debug logging only for *specified client addresses*:

```
error_log /path/to/log;

events {
  debug_connection 192.168.1.1;
  debug_connection 192.168.10.0/24;
}
```

4.1.2 Cyclic Memory Buffer

Debug log can be written to a cyclic memory buffer:

```
error_log memory:32m debug;
```

Writing to the memory buffer at the `debug` level will not significantly impact performance even under high load. In this case, the log can be extracted using a GDB script, for example:

```
set $log = ngx_cycle->log

while $log->writer != ngx_log_memory_writer
  set $log = $log->next
end

set $buf = (ngx_log_memory_buf_t *) $log->wdata
dump binary memory debug_log.txt $buf->start $buf->end
```

CHAPTER 5

Intellectual Property Rights

The documentation for the software product Angie PRO is the intellectual property of Web Server, LLC. The documentation was created as a result of modifications (revisions) to the documentation for the software product Nginx.

Index

G

H

I

J

K

L

M

O

P

Q

R

S

T

U

V

W

X

Z