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

User Guide

version 1.11.8

Web Server, LLC

Jun 19, 2026

Contents

og:description
: Information for operating Angie PRO software

CHAPTER 1

Annotation

This document contains information necessary for operating 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 probes 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, references, indexes, and instructions for configuring Angie.

3.1 General Information

These articles cover installation and configuration of Angie, starting and stopping the web server, managing it, as well as various aspects of request processing 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 Runtime 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 disk before exiting, so the new master process can restore them and thereby improve performance. If the restore fails due to a change in zone size, binary version incompatibility, or other reasons, Angie will log a warning (`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 v1.11.8 #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 v1.11.8 #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 disk before upgrading, so the new master process can restore them and thereby improve performance. If the restore fails due to a change in zone size, binary version incompatibility, or other reasons, Angie will log a warning (`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*	Set *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*	Use the specified *prefix* path for `angie` (the directory where server files are located; the default is `/usr/local/angie/`).
`-q`	Display only error messages if `-t` or `-T` is set; otherwise, has no effect.
`-s` *signal*	Send a *signal* to the master process: `stop`, `quit`, `reopen`, `reload`, and so on.
`-t`	Test the configuration file, then exit. Angie checks the configuration syntax, recursively including 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, 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 probes. 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 `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;
```

3.2 References and Indexes

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

3.2.1 Native Modules

This guide describes Angie's native modules, 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:;
    #  ...
}
```

Note

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*] [[`filter=`*type*:*value*] ...];
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.

Optional `filter=` parameters restrict which messages are written to the log. Supported filters are:

- `filter=file:`*prefix* matches a source file prefix (for example, `ngx_http_access_module.c`);
- `filter=event:`*prefix* matches an event ID prefix (for example, `http.upstream.peer`);
- `filter=tag:`*tag* matches a tag attached to the log entry.

Multiple `filter=file:` or `filter=event:` parameters are matched by prefix, and any match allows the message to pass. Multiple `filter=tag:` parameters require all tags to be present. Tags can be added automatically by modules (for example, `http`, `stream`, `mail`, `upstream`, `peer`, `subrequest`) and by *error_log_user_tag* directives.

Note

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

Each entry in the error log has the following format:

```
timestamp [level] PID#TID: *request_id message
```

Where:

- `timestamp` — date and time of the event
- `level` — logging level of the event
- `PID#TID` — process and thread identifiers
- `*request_id` — unique request identifier (if applicable)
- `message` — error or event message text

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;
```

If a dynamic module was built for a different Angie build, loading fails with an error like: "module "..." was built for "..." but you are running "Angie"".

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.

Note

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 reading and sending of files *without blocking* worker processes.

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

If all threads in the pool are busy executing tasks, new tasks wait in a queue. The `max_queue` parameter limits the number of tasks allowed to be waiting in the queue. By default, up to 65536 tasks can be in the queue. When the queue overflows, the task is completed with an error.

timer_resolution

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

Reduces timer resolution in worker processes, thus reducing the number of `gettimeofday()` system calls. By default, `gettimeofday()` is called each time a kernel event is received. With reduced resolution, `gettimeofday()` is only called once per specified interval.

Example:

```
timer_resolution 100ms;
```

Internal implementation of the interval depends on the method used:

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

use

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

Specifies the *method* to use for *connection processing*. There is normally no need to specify it explicitly, because Angie will by default use the most efficient method.

user

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

Defines *user* and *group* credentials used by worker processes (see also build parameters). If *group* is omitted, a group whose name equals that of *user* is used.

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 that can be opened by a worker process.

It should be kept in mind that this number includes all connections (e.g. connections with proxied servers, among others), not only connections with clients. Another consideration is that the actual number of simultaneous connections cannot exceed the current limit on the maximum number of open files, which can be changed by *worker_rlimit_nofile*.

worker_cpu_affinity

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

Binds worker processes to the sets of CPUs. Each CPU set is represented by a bitmask of allowed CPUs. There should be a separate set defined for each of the worker processes. 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 binding worker processes automatically 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;
```

Note

The 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 like it is done by the `nice` command: a negative *number* means higher priority. Allowed range normally 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 many factors including (but not limited to) the number of CPU cores, the number of hard disk drives that store data, and load pattern. When one is in doubt, setting it to the number of available CPU cores would be a good start (the value `"auto"` will try to autodetect it).

worker_rlimit_core

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

Changes the limit on the largest size of a core file (`RLIMIT_CORE`) for worker processes. Used to increase 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. Used to increase the limit without restarting the main process.

worker_shutdown_timeout

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

Configures a timeout in seconds for a graceful shutdown of worker processes. When the specified time expires, Angie will try to close all the connections currently open to facilitate shutdown.

Graceful shutdown is initiated by sending a *QUIT signal* to the main process, which instructs worker processes to stop accepting new connections and allows existing connections to complete. Worker processes continue to handle active requests until they finish, then shut down gracefully. If connections remain open longer than `worker_shutdown_timeout`, Angie will forcibly close these connections to complete the shutdown. Also, client keep-alive connections are closed only if they have been idle for at least the time specified by *lingering_timeout*.

working_directory

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

Defines the current working directory for a worker process. It is primarily used when writing a core file, in which case a worker process should have write permission for the specified directory.

HTTP Module

Access

The module controls access to server resources based on client IP addresses or networks. It allows permitting or blocking access for specific IP addresses, 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 IP addresses.

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 IP addresses.

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, the module isn't built by default; it must be enabled with the build option `--with-http_acme_module`. In packages and images from our repositories, the module is included in the build.

Configuration Example

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

Directives

acme

Changed in version 1.9.0: The "no valid domain name defined for ACME client" error is issued only if no valid (i.e. ACME-compliant) domain name is found in the *server* block that references the ACME client.

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

Specifies the *ACME client* that obtains a certificate for the domains in this *server* block. A single certificate covers all domains specified in the *server_name* directives of every *server* block that references the client with the given *name*; if the `server_name` configuration changes, the certificate is renewed to reflect the changes.

Each time Angie starts, new certificates are requested for all domains that are missing a valid certificate. Possible reasons include certificate expiration, missing or unreadable files, and changes in certificate settings.

Note

This directive only controls which domain names are included in certificate requests; it does not affect where the certificate can be used. Any `server` block can reference the certificate through the *$acme_cert_<name>* variable, regardless of whether the block contains an `acme` directive. Removing `acme` from a `server` block simply excludes that block's *server_name* values from future certificate requests, but does not prevent the block from using the certificate.

Note

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

Wildcard domains are supported only with `challenge=dns` 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

Changed in version 1.9.0.

Changed in version 1.11.0.

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

Defines an ACME client with a globally unique *name*. It must be valid for a directory, is a string with variables, and will be used case-insensitively.

Each client manages a single certificate; to obtain separate certificates, configure multiple `acme_client` blocks (see *Separate Certificates for Different Domains*).

Tip

The client name specified 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 Let's Encrypt ACME directory URI is specified as https://acme-v02.api.letsencrypt.org/directory.

Note

The ACME module adds a named `location @acme` to the *client* context, which can be used to configure requests to the ACME directory; by default, this `location` contains a *proxy_pass* directive with the directory *uri*, to which other settings from the *Proxy* module can be added.

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, the Let's Encrypt staging environment is 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 the new certificate in shared memory; the more domains the certificate is requested for, the more space is required. This parameter does not limit the size of ACME server responses; use *acme_max_response_size* for that. If the parameter is not set, Angie calculates an approximate size based on the configured domain list and uses it for shared memory allocation. If a certificate already exists at startup but its size exceeds the `max_cert_size` value, the `max_cert_size` value is dynamically increased to match the size of the existing certificate file. If the size of a certificate obtained during renewal exceeds `max_cert_size`, the renewal process will fail with an error. Default: calculated automatically.
`max_key_auth_siz`	Limits the size of the key authorization string that Angie stores in shared memory for an ACME challenge. If the ACME server returns a key authorization string larger than this value, the request fails with an error that advises raising `max_key_auth_size`. Although specified on the `acme_client` line, this is a single setting shared by all clients in the `http` block. Default: `2k`.
`renew_before_exp`	*Time* before certificate expiration when renewal should begin. Default: `30d`.
`renew_on_load`	Specifies that the certificate should be forcibly renewed each time the configuration is loaded.
`retry_after_erro`	*Time* to wait before retrying if certificate retrieval failed. If set to `off`, the client will not retry to obtain the certificate after an error. Default: `2h`.
`challenge`	Specifies the verification type for the ACME client. Valid values: `dns`, `http`, `alpn`. The `alpn` value enables TLS-ALPN-01 validation and requires Angie to be built with OpenSSL that supports ALPN (not supported with BoringSSL or AWS-LC builds). 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 automatic generation, 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, ensure that 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_max_response_size

Added in version 1.11.0.

Syntax	`acme_max_response_size` *size*;
Default	`acme_max_response_size 32k;`
Context	http

Limits the maximum size of an ACME server response body. If a response exceeds this limit, the request fails with an error. Increase the value if you see errors like `too big subrequest response while sending to client`.

acme_client_path

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

Overrides the *path* to the directory for storing certificates and keys, set during build using the build parameter `--http-acme-client-path`.

acme_dns_port

Changed in version 1.9.1.

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 in the range from 1 to 65535.

Specifying an IP address along with an 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 port number 1024 or lower, Angie must run with *superuser* privileges.

acme_http_port

Added in version 1.11.0.

Changed in version 1.11.1.

Syntax	`acme_http_port` *port* | *ip*[:*port*] | *[ip6]*[:*port*];
Default	`acme_http_port 80;`
Context	http

Specifies the port that the module uses to handle HTTP ACME challenge requests. The port number must be in the range from 1 to 65535.

Specifying an IP address along with an 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_http_port 8080;
acme_http_port 127.0.0.1;
acme_http_port [::1];
```

If no server is configured to listen on the specified address and port, the module creates a dedicated listener for HTTP challenges.

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

acme_hook

Changed in version 1.9.0.

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

Enables hook-based domain validation for the *ACME client* specified by *name*. When certificate issuance or renewal requires domain verification, Angie generates an internal request to the named `location` where this directive is placed. How the request is handled depends entirely on the other directives configured in the same `location`, such as *fastcgi_pass*, *proxy_pass*, or any other request handler.

name	The name of the *ACME client* for which this hook handles domain verification.
uri	A string with variables; specifies the request URI for hook calls. Default: `/`.

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

```
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*.

Note

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

`$acme_hook_challenge`

The challenge type. Possible values: `dns`, `http`, `alpn`.

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

`$acme_hook_name`

The hook name. For different challenge types, it may have different values and meanings:

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

`$acme_hook_token`

The verification token. For HTTP challenge, 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 `API` module implements an HTTP RESTful interface for obtaining basic information about the web server in JSON format, as well as *statistics* on client connections, shared memory zones, DNS queries, HTTP requests, HTTP response cache, *stream* module sessions, and zones of the *limit_conn http*, *limit_conn stream*, *limit_req*, and *http upstream* modules.

The interface accepts `GET` and `HEAD` HTTP methods; a request with another method will cause an error:

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

In Angie PRO, this interface includes a *dynamic configuration* section that allows changing settings without reloading the configuration or restarting; currently, configuration of individual servers within *upstream* is available.

Directives

api

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

Enables the HTTP RESTful interface in `location`.

The *path* parameter is mandatory. Similar to the *alias* directive, it sets the path for replacing the one specified in `location`, but over the API tree rather than the filesystem.

If specified in a prefix `location`:

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

the part of the request URI matching the prefix */stats/* will be replaced with the path specified in the *path* parameter: */status/http/server_zones/*. For example, a request to */stats/foo/* will access the API element `/status/http/server_zones/foo/`.

Variables are allowed: *api /status/$module/server_zones/$name/* and usage inside regex location:

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

Here the *path* parameter defines the full path to the API element; thus, from a request to `/api/location/data/` the following variables will be extracted:

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

And the final request will be `/status/http/location_zones/data/`.

Note

In Angie PRO, you can separate the *dynamic configuration API* and the immutable *status API* that reflects the current state:

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

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

The *path* parameter also allows controlling API access:

```
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;
}
```

Note

If `api` is placed in a `location` with a trailing slash in the prefix (for example, `location /name/`), and the *auto_redirect* directive is set to `default`, requests without a trailing slash will be redirected (`/name -> /name/`).

api_config_files

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

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

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

A request to `/status/angie/` returns approximately the following:

```
{
    "version":"1.11.8",
    "address":"192.168.16.5",
    "generation":1,
    "load_time":"2026-06-18T12:58:39.789Z",
    "config_files": {
        "/etc/angie/angie.conf": "...",
        "/etc/angie/mime.types": "..."
    }
}
```

By default, output is disabled because configuration files may contain particularly sensitive, confidential information.

Metrics

Angie publishes usage statistics in the `/status/` API section; you can open access to it by setting the appropriate `location`. Full access:

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

Example of partial access, already shown above:

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

Example configuration

With a configuration including `location /status/`, `resolver`, `http` in `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;
        proxy_cache_valid 200 10m;

        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;
        }

    }
}
```

In response to the request `curl https://www.example.com/status/`, Angie returns:

JSON tree


```
{
    "angie": {
        "version":"1.11.8",
        "address":"192.168.16.5",
        "generation":1,
        "load_time":"2026-06-18T12:58:39.789Z"
    },

    "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
            }
        }
    }
}
```


A set of metrics can be requested by individual JSON branch by constructing the appropriate request. For example:

```
$ 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/acme_clients
$ curl https://www.example.com/status/http/acme_clients/<client>
$ curl https://www.example.com/status/http/metric_zones
$ curl https://www.example.com/status/http/metric_zones/<zone>/metrics
$ 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 format strings for dates; to use the integer UNIX 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`

Changed in version 1.9.0: The `build_time` field was added.

```
{
    "version": "1.11.8",
    "build_time": "2026-06-18T16:05:43.805Z",
    "address": "192.168.16.5",
    "generation": 1,
    "load_time": "2026-06-18T16:15:43.805Z"
    "config_files": {
        "/etc/angie/angie.conf": "...",
        "/etc/angie/mime.types": "..."
    }
}
```

`version`	String; version of the running Angie web server
`build`	String; particular build name if 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 the API request
`generation`	Number; total number of configuration reloads since last start
`load_time`	String; time of the last configuration reload in the *date* format; string values 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" }` **Warning** The `config_files` object is available in `/status/angie/` only if the *api_config_files* directive is enabled.

`/status/angie/license` (PRO)

Added in version 1.11.0: PRO

```
{
    "path": "/etc/angie/license.pem",
    "status": "valid",
    "owner": "Example Corp",
    "days_left": 30,
    "since": "2026-01-01",
    "until": "2027-01-01",
    "limits": {
        "worker_processes": 16,
        "worker_connections": 65535
    }
}
```

`path`	String; full path to the license file
`status`	String; license status: `missing`, `invalid`, `valid`, `grace`, `expired`, or `pending`
`owner`	String; license owner from the certificate subject
`days_left`	Number; days until the license changes state. A negative value means the license has expired, and the value is the number of days since expiration
`since`	String; license validity start date
`until`	String; license validity end date
`limits`	Object; licensed limits for the current instance

Connections

`/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

Shared memory zones with slab allocation

`/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;
proxy_cache_valid 200 10m;
```

The specified shared memory zone will collect the following statistics:

`pages`	Object; memory pages statistics
:samp: *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 data for memory slot sizes (`8`, `16`, `32`, etc., up to half of the page size in bytes)
`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 of specified size
`fails`	Number; the number of unsuccessful attempts to allocate memory of specified size

Example:

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

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

DNS queries to resolver

`/status/resolvers/<zone>`

To collect resolver statistics, the *resolver* directive must set the `status_zone` parameter (*HTTP* or *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 of responses with code 1 (Format Error)
`server_failure`	Number; the number of responses with code 2 (Server Failure)
`not_found`	Number; the number of responses with code 3 (Name Error)
`unimplemented`	Number; the number of responses with code 4 (Not Implemented)
`refused`	Number; the number of 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

Note

`queries` and `responses` count every resolution request Angie makes internally, including those served from the TTL cache. `sent` counts packets actually dispatched to the name server; the gap between the two reflects cache hits.

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
  }
}
```

`/status/http/metric_zones/<zone>`

Custom metrics defined by *metric_zone* or *metric_complex_zone* in the `http` context. Metrics are updated with the *metric* directive or the module variables.

`discarded`	Number; the number of metric entries discarded because the zone ran out of memory.
`metrics`	Object; metrics per key. For single-metric zones, values are numbers. For complex zones, values are objects with metric names. For histogram mode, values are objects with bucket names.

If `discard_key` is set and some entries have been expired, their aggregated metrics are exposed under this key.

Example:

```
{
    "discarded": 3,
    "metrics": {
        "example.com": {
            "count": 42,
            "max": 8
        }
        "expired": {
            "count": 10,
            "max": 3.2
        }
    }
}
```


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;
proxy_cache_valid 200 10m;
```

/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 stale 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 stale 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

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
        }
    }
}
```

ACME clients

`/status/http/acme_clients/<client>`

For each configured *acme_client* in the `http` block, returns the current client and certificate status:

```
{
  "state": "ready",
  "certificate": "valid",
  "details": "The client is ready to request a certificate.",
  "next_run": "2026-06-18T16:15:43.805Z"
}
```

`state`	String; ACME client state. Possible values: `ready`, `requesting`, `disabled`, `failed`.
`certificate`	String; certificate status. Possible values: `valid`, `expired`, `missing`, `mismatch`, `error`.
`details`	String; short status details from the last ACME operation.
`next_run`	Date; next scheduled attempt to request or renew the certificate. Not returned when `state` is `disabled` or `requesting`.

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 requests
`skipped`	Number; the total number of requests passed with zero-length key, or key exceeding 255 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

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>`

Changed in version 1.9.0: The `busy` peer state was added.

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
`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 to it; • `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 the peer
`total`	Number; total number of requests forwarded to the peer
`last`	String or number; time when the peer was last selected, formatted as a *date*
`max_conns`	Number; the configured *maximum* number of simultaneous active connections to the peer, if specified
`responses`	Object; response 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 the peer
`sent`	Number; the total number of bytes sent to the peer
`health`	Object; health statistics
`fails`	Number; the total number of unsuccessful attempts to communicate with the peer
`unavailable`	Number; how many times the peer became `unavailable` due to reaching the *max_fails* limit
`downtime`	Number; the total time (in milliseconds) when the peer was `unavailable` for selection
`downstart`	String or number; time when the peer became `unavailable`, formatted as a *date*. This field is present only while the peer is in the `unavailable` state; otherwise it is absent
`header_time`	Number; average time (in milliseconds) to receive the response headers from the server; see *response_time_factor (PRO)*
`response_time`	Number; average time (in milliseconds) to receive the entire response from the server; see *response_time_factor (PRO)*
`sid`	String; *configured id* of the server in the upstream group
`keepalive`	Number; the current number of 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; the level of the active group that is currently used for load balancing requests. If the active group is the primary one, the value is 0
`timeout`	Number; remaining wait time in milliseconds, after which the balancer will re-check for healthy nodes in groups with lower levels, starting from the primary group, while groups with higher levels are not checked; not displayed for the primary group (level 0)

`health/probes` (PRO)

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

```
{
    "192.168.16.4:80": {
        "state": "unhealthy",
        "...": "...",
        "health": {
            "...": "...",
            "probes": {
                "count": 10,
                "fails": 10,
                "last": "2026-06-18T09:56:07Z"
            }
        }
    }
}
```

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

Counters in `probes`:

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

`queue` (PRO)

If a *request queue* is configured for the upstream, the upstream object also contains a nested `queue` object with request queue counters:

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

Counter values are summed across all worker processes:

`queued`	Number; total number of requests that entered the queue
`waiting`	Number; current number of requests in the queue
`dropped`	Number; total number of requests removed from the queue because the client prematurely closed the connection
`timedout`	Number; total number of requests removed from the queue due to timeout
`overflows`	Number; total number 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>`

Changed in version 1.9.0: The `busy` peer state was added.

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 the *server* directive
`backup`	Boolean; `true` for backup servers
`weight`	Number; the *weight* set for 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 to it • `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; statistics on selecting this peer for connections
`current`	Number; current number of connections to the peer
`total`	Number; total number of 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 statistics
`received`	Number; total bytes received from the peer
`sent`	Number; total bytes sent to the peer
`health`	Object; peer health statistics
`fails`	Number; total failed attempts to reach the peer
`unavailable`	Number; total number of times the peer became `unavailable` due to reaching the *max_fails* value
`downtime`	Number; total time (in milliseconds) that the peer was `unavailable` (unavailable for selection)
`downstart`	String or number; time when the peer last became `unavailable`, formatted as a *date*. This field is present only while the peer is in the `unavailable` state; otherwise it is absent
`connect_time`	Number; average time (in milliseconds) to establish a connection with the server; see the *response_time_factor (PRO)* directive
`first_byte_time`	Number; average time (in milliseconds) to receive the first byte of the response from the server; see the *response_time_factor (PRO)* directive
`last_byte_time`	Number; average time (in milliseconds) to receive the complete response from the server; see the *response_time_factor (PRO)* directive
`backup_switch` (PRO 1.10.0+)	Object; contains the current state of active backup logic, present if *backup_switch (PRO)* is configured for the upstream
`active`	Number; level of the active group currently used for load balancing. If the active group is the primary group, the value is 0
`timeout`	Number; remaining wait time in milliseconds after which the load balancer will recheck for healthy nodes in groups with lower levels, starting from the primary group, while groups with higher levels are not checked; not displayed for the primary group (level 0)

In Angie PRO, if the upstream has *upstream_probe (PRO)* probes configured, the `health` object also

has a `probes` subobject that stores the server's health probe counters, while `state`, in addition to the values from the table above, can also be `checking` and `unhealthy`:

```
{
    "192.168.16.4:80": {
        "state": "unhealthy",
        "...": "...",
        "health": {
            "...": "...",
            "probes": {
                "count": 2,
                "fails": 2,
                "last": "2026-06-18T11:03:54Z"
            }
        }
    }
}
```

The `checking` value of `state` means that the server, which has a probe configured with the `essential` parameter, hasn't been checked yet; the `unhealthy` value means that the server is non-operational. Both states also mean that the server isn't included in load balancing. For details of health probes, see *upstream_probe*.

Counters in `probes`:

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

Dynamic Configuration API (PRO)

The API includes a `/config` section that enables dynamic updates to Angie's configuration in JSON format 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`, `slow_start`, `backup`, `down` and `sid` parameters, as described in *server*.

Note

There is no separate `drain` (PRO) parameter 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,
    "slow_start": 0,
    "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 with the `random` method:

```
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 when adding a new peer.

`/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` block; 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`, `slow_start`, `backup`, and `down` parameters, as described in *server*.

Note

There is no separate `drain` (PRO) parameter here; to enable `drain` mode, 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,
    "slow_start": 0,
```

```
    "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 with the `random` method:

```
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 when adding a new peer.

When deleting peers, 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 each HTTP method applicable to this section using the following upstream configuration as an example:

```
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 the `defaults=on` argument:

```
$ 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,
        "slow_start": 0,
        "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 add the `max_fails` parameter, not specified earlier, to 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; in doing so, it restores the default values if there are any.

For example, to delete the previously modified `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 the `defaults=on` argument:

```
$ 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,
    "slow_start": 0,
    "backup": false,
    "down": false,
    "sid": ""
}
```

The `max_fails` parameter has returned 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` parameter 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
}
```

Note that the JSON object supplied with the `PATCH` request *was merged* with the existing one instead of replacing it entirely, as would be the case with `PUT`.

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

Note

This deletion is identical to `DELETE`; in particular, it restores the default values.

For example, to delete the `down` parameter 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` value was supplied, was deleted; the `max_conns` value 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. The format is as follows:

```
# 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).

Warning

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

Serves requests ending with a slash (`/`) and produces a directory listing. Usually, a request is passed to the `AutoIndex` module when the *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 *XSLT* module.

Output Formats

Object fields in responses contain the following data:

Field	Description
`name`	File or directory name
`type`	Object type: `file` or `directory`
`size`	Object size according to *autoindex_exact_size*; for directories — `0`
`mtime`	Last modification time in Unix time format

HTML

```
<html>
<head>
    <title>Index of /files/</title>
</head>
<body>
    <h1>Index of /files/</h1>
    <hr>
    <pre>
            <a href="../">../</a>
            <a href="example.txt">example.txt</a>                12-Jun-2025 14:21
↪1234
            <a href="image.png">image.png</a>                    12-Jun-2025 14:21
↪4321
            </pre>
    <hr>
</body>
</html>
```

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<listing>
<file>
    <name>example.txt</name>
    <type>file</type>
    <size>1234</size>
    <mtime>2025-06-12T14:21:00Z</mtime>
</file>
<file>
    <name>image.png</name>
    <type>file</type>
    <size>4321</size>
```

```
    <mtime>2025-06-12T14:21:00Z</mtime>
</file>
</listing>
```

JSON

```
[
{
    "name": "example.txt",
    "type": "file",
    "size": 1234,
    "mtime": "2025-06-12T14:21:00Z"
},
{
    "name": "image.png",
    "type": "file",
    "size": 4321,
    "mtime": "2025-06-12T14:21:00Z"
}
]
```

JSONP

```
callback([
{
    "name": "example.txt",
    "type": "file",
    "size": 1234,
    "mtime": "2025-06-12T14:21:00Z"
},
{
    "name": "image.png",
    "type": "file",
    "size": 4321,
    "mtime": "2025-06-12T14:21:00Z"
}
]);
```

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* variable.

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 formats 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* variable.

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 responses received from a proxied or a FastCGI/uwsgi/SCGI/gRPC server when the responses 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` *charset*;
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.

Note

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
```

Docker

Added in version 1.10.0.

The module provides dynamic configuration of proxied server groups in both *HTTP* and *stream* contexts based on Docker container labels. For the functionality to work, a shared memory zone must be configured in the group (see the `zone` description for *http* and *stream*).

Note

The module supports working with both Docker and its alternatives, such as Podman, which implement a compatible API. The recommended Podman version is 4.9.3 or higher.

The module connects to the Docker daemon via API, the interaction method with which is specified by the *docker_endpoint* directive. After obtaining a list of running containers, Angie analyzes them for the presence of suitable *labels*. If a container description contains a label with a port, then the address and port of such a container, as well as parameters from other labels of this container, are automatically added to the corresponding `upstream` block in the Angie configuration.

Note

The same container can be added to multiple `upstream` groups. To do this, simply specify multiple sets of labels with different group names and ports.

This is especially useful if the container runs several different services on different ports — each service can be associated with its own group.

The module then subscribes to container lifecycle events and begins updating the proxied server configuration without reloading Angie:

- when starting a container with suitable labels, its internal IP address is added to the specified group;
- when stopping or removing a container, it is automatically removed from the group;
- when pausing a container with the `docker pause` command, the server is marked as `down`, and with `docker unpause` — as `up`.

Configuration Example

The module's directives are always located in the `http` context, but proxied server groups can be defined in both the `http` context and the `stream` context.

Configuration example for `http`:

```
http {

    # Examples of connection options:
    # docker_endpoint http://127.0.0.1:2375;
    # docker_endpoint https://127.0.0.1:2376;
    docker_endpoint unix:/var/run/docker.sock;

    # maximum Docker response buffer size (optional)
    # docker_max_object_size 128k;

    upstream u {

        zone z 1m; # shared memory zone is required
    }

    server {

        listen 80;
        server_name example.com;

        location / {

            proxy_pass http://u;
        }
    }
}
```

Similarly in stream context:

```
http {

    # Examples of connection options:
    # docker_endpoint http://127.0.0.1:2375;
    # docker_endpoint https://127.0.0.1:2376;
    docker_endpoint unix:/var/run/docker.sock;

    # maximum Docker response buffer size (optional)
    # docker_max_object_size 128k;
}

stream {

    upstream u {

        zone z 1m;
    }

    server {

        listen 12345;
        proxy_pass u;
    }
```

```
}
```

Upon receiving an event for a container, Angie looks for labels of the form `angie.http.upstreams.<name>.port=<port>` (for HTTP context) or `angie.stream.upstreams.<name>.port=<port>` (for stream context). When a label is present, the container's address in the specified Docker network (or the first available one if the `angie.network` label is not specified) is added to the corresponding proxied server group.

If a container stops or is removed, the server is removed from the group; if a container is paused, the server is marked as `down`.

Fragment of a `docker-compose.yml` file with labels that Angie recognizes:

```
services:
  myapp:
    image: myapp:latest
    labels:
      - "angie.http.upstreams.u.port=8080"
      - "angie.network=my_bridge"
      - "angie.http.upstreams.u.weight=2"
      - "angie.http.upstreams.u.max_conns=50"
      - "angie.http.upstreams.u.max_fails=3"
      - "angie.http.upstreams.u.fail_timeout=10s"
      - "angie.http.upstreams.u.backup=true"
```

Labels

Labels specify server parameters in the proxied server group similar to the arguments of the `server` directive:

Label	Purpose
`angie.(http|stream).upstreams.<name>.port=<port>` *(required)*	Container port that Angie will connect to; the container itself is added to the group named `<name>`.
`angie.network=<docker-network>`	Name of the Docker network from which to take the container's IP address.
`angie.(http|stream).upstreams.<name>.weight=<n>`	Value of the `weight` parameter.
`angie.(http|stream).upstreams.<name>.max_conns=<n>`	Maximum number of simultaneous connections (`max_conns`).
`angie.(http|stream).upstreams.<name>.max_fails=<n>`	Threshold for failed attempts (`max_fails`).
`angie.(http|stream).upstreams.<name>.fail_timeout=<t>`	Interval for counting failed attempts (`fail_timeout`).
`angie.(http|stream).upstreams.<name>.backup=true|false`	Marks the server as `backup`.
`angie.(http|stream).upstreams.<name>.sid=<string>`	Sets a custom server identifier (`sid`) for the proxied server.
`angie.(http|stream).upstreams.<name>.slow_start=<time>`	Enables `slow_start` mode with a configurable time period.

Directives

docker_endpoint

Syntax	`docker_endpoint URL;`
Default	—
Context	http

Specifies the method of connecting to the Docker daemon and enables tracking of container events. The following options are supported:

`unix:/var/run/docker.sock`	Connection via Unix socket (e.g., `/var/run/docker.sock`).
`http://host:port`, `https://host:port`	Connection via HTTP or HTTPS to a remote Docker API.

The connection can be additionally configured using the *client* context, where the module adds two named `location` blocks:

- `@docker_events` is used to receive container events;
- `@docker_containers` — to get container information.

By default, they contain the *proxy_pass* directive with the connection address and several other optimal default settings, to which other settings from the *Proxy* module can be added.

If the directive is specified, Angie opens a connection to Docker using the specified method, requests a list of running containers, analyzes their labels and processes all subsequent container events, adding or removing servers in proxied server groups according to the labels.

Tip

To access the Docker daemon via Unix socket (`/var/run/docker.sock` or another), the *user* which Angie runs as must have read and write permissions for this socket.

docker_max_object_size

Syntax	`docker_max_object_size <size>;`
Default	`64k`
Context	http

Sets the maximum buffer size that is used for both JSON responses to Docker requests and for the container event stream.

- For regular requests (API version, container list, container information): the entire response must fit in the buffer, otherwise an error occurs.
- For container events, streaming processing is used with buffer reuse, which allows processing an unlimited stream of events.

The typical value of `64k` is sufficient for approximately 25 containers.

When Docker API connection errors or response processing errors occur, the module automatically retries at specific time intervals. The maximum number of retry attempts for getting information about

a specific container is limited to two *additional* attempts; after that, the module stops attempting for that container.

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

Enables emitting a single-pixel transparent GIF in the containing 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.

Note

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.

`on`	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. Sending to the client is performed in parallel: filled buffers are passed for sending, but their total size is limited by *fastcgi_busy_buffers_size*. If a buffer is not completely filled, it is not passed for sending unless it contains the last part of the response. Therefore, buffered reading is not suitable when you need immediate delivery of every few bytes. 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.
`off`	The response is passed to a client immediately as it is received. Angie works in a "read — send" loop and does not wait for the buffer to fill completely: for example, 10 bytes read from a 4K buffer are sent right away. At the same time, if the entire response fits into the buffer, Angie can read it in full. The maximum size of the data that Angie can receive from the server at a time is set by the *fastcgi_buffer_size* directive. With `off`, *fastcgi_limit_rate* does not work.

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 together 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 passed to the FastCGI server for populating a new cache element has not completed for the specified time, one more request may be passed 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.

Warning

Cache metadata is stored in shared memory. Manually deleting cache files does not reset the counters and may lead to unpredictable behavior. To completely reset the cache, stop the server, delete the cache directory, and start again.

Note

Third-party cache purge modules (e.g., Cache Purge) only delete files but do not reset the *fastcgi_cache_min_uses* counter. The directive is intended to protect the cache from pollution by infrequent requests, and resetting the counter on purge may negatively affect performance.

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. Cache metadata is stored in shared memory.

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 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`	maximum number of cache items to load in one iteration Default: `100`
`loader_threshold`	limits the time of one iteration Default: `200` milliseconds
`loader_sleep`	time for which a pause is maintained between iterations 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 cannot be selected for processing a request.
`updating`	an additional parameter that 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.

The use of a stale cached response can also be permitted 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 method has lower priority than setting 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, it can be specified to cache any responses using the `any` parameter:

```
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 proxied server. The *rate* is specified in bytes per second; variables can be used.

`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 entire 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.

Note

In the standard `fastcgi.conf` and `fastcgi_params` files shipped with Angie, `REQUEST_METHOD` is set to `$upstream_request_method`. This ensures that when caching converts `HEAD` to `GET`, the upstream request method reflects that conversion.

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*.

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*.

Note

If `fastcgi_pass` is placed in a `location` whose prefix ends with a slash (for example, `location /name/`), and the *auto_redirect* directive is set to `default`, requests without a trailing slash will be redirected (`/name -> /name/`).

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	`fastcgi_pass_request_body on;`
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	`fastcgi_pass_request_headers on;`
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.

`off`	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 parameter)
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 under 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 run as FastCGI servers, these parameters are usually available as environment variables. For example, the `User-Agent` header field is passed as the HTTP_USER_AGENT parameter. Besides HTTP request header fields, arbitrary parameters can be passed 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 are used, in particular, to 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 group set by this directive.

`$fastcgi_path_info`

The value of the second capture group 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.
`volatile`	indicates that the variable is not cacheable.

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.

Note

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` *address* | *CIDR* | *unix:*;
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.

Note

This module requires the *HTTP2* module.

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.

Note

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* `DELETE` command.

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 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.

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* group:

`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 the 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*.

Note

If `grpc_pass` is specified in a `location` with a trailing slash in the prefix (for example, `location /name/`), and the *auto_redirect* directive is set to `default`, requests without a trailing slash will be redirected (`/name -> /name/`).

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 the 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.

`off`	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*.

The value `store:scheme:id` can be specified instead of the file, which is used to load a secret key with a specified *id* and OpenSSL provider registered URI *scheme*, such as pkcs11.

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.

Warning

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.

Note

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.

Warning

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` name 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 name from *grpc_pass* 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

Enables the specified protocols for requests to a gRPC SSL 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

A filter that compresses responses using the gzip method, which allows reducing the size of transmitted data by 2 times or more.

Warning

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 (1.3.10), 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, WebP, HEIC, and AVIF 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`.

Note

This module utilizes the libgd library. It is recommended to use the latest available version of the library.

To transform images in WebP, HEIC, or AVIF formats, the `libgd` library must be compiled with support for these formats.

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

Changed in version 1.11.0.

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*; • `image_filter` convert *type*;
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 JPEG, GIF, PNG, WebP, HEIC, or AVIF 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.
`convert` *type*	converts an image to the specified output format. Valid values are `jpeg`, `gif`, `png`, `webp`, `heic`, and `avif`. The parameter value can contain variables.

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.

image_filter_heic_quality

Syntax	`image_filter_heic_quality` *quality*;
Default	`image_filter_heic_quality 80;`
Context	http, server, location

Sets the desired quality of the transformed HEIC images. Acceptable values are positive numbers. Parameter value can contain variables.

image_filter_avif_quality

Syntax	`image_filter_avif_quality` *quality* [*speed*];
Default	`image_filter_avif_quality 80 6;`
Context	http, server, location

Sets the desired quality of the transformed AVIF images. The optional `speed` parameter controls the encoder speed; both values must be positive numbers. Parameter values 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".

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 HTTP/3, 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=`*name* [`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.

Logs are written in the context of a `location` where processing ends. This may be a `location` different from the original one if an *internal redirect* occurs 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;` (path depends on the build parameter `--http-log-path`)
Context	http, server, location, if in location, limit_except

Sets the *path*, *format* and buffered write settings for the log. Multiple logs can be used at the same configuration level. Logging to *syslog* is configured by specifying the *"syslog:"* prefix in the first parameter. The special value *off* cancels all *access_log* directives for the current level. If no format is specified, the predefined *"combined"* format is used.

If the buffer size is specified using the `buffer` parameter or the `gzip` parameter is specified, writing will be buffered.

Warning

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, data is written to the file:

- if the next log line does not fit in the buffer;
- if the data in the buffer has been there longer than the time interval specified by the *flush* parameter;
- when *reopening the log file* or terminating the worker process.

If the `gzip` parameter is specified, the buffer will be compressed before writing to the file. The compression level can be set in the range from 1 (faster, but worse compression) to 9 (slower, but better compression). By default, a buffer size of *64K* bytes and compression level *1* are used. Data is compressed in atomic blocks, and at any time the log file can be decompressed or read using the `zcat` utility.

Example:

```
access_log /path/to/log.gz combined gzip flush=5m;
```

Note

For gzip compression of logs support, Angie must be built with the *zlib* library.

Variables can be used in the file path, but such logs have some limitations:

- the *user* under whose credentials the worker processes run must have permissions to create files in the directory with such logs;
- buffering does not work;
- the file is opened for each log write and closed immediately after writing. However, since descriptors of frequently used files can be stored in cache, during log rotation within the time specified by the *valid* parameter of the *open_log_file_cache* directive, writing may continue to the old file.
- for each log write, the existence of the root directory for the request is checked — if this directory does not exist, the log is not created. Therefore *root* and *access_log* should be described at 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 evaluation result is `"0"` or an empty string. In the following example, 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`=:samp:*default* | `json` | `none`] *string* ...;
Default	`log_format combined "...";`
Context	http

Specifies the log format.

The `escape` parameter allows setting character escaping to `json` or `default` in variables; by default `default` is used. The `none` value disables character escaping.

When using `default`, characters `"`, `\`, and characters with values less than 32 or greater than 126 are escaped as `\xXX`. If the variable value is not found, a hyphen `-` will be written to the log as the value.

When using `json`, all characters not allowed in JSON strings are 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 the client start with the prefix `sent_http_`, for example, `$sent_http_content_range`.

The predefined format `combined` always exists in the configuration:

```
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 file descriptors of frequently used logs whose names are specified using variables. Parameters:

`max`	Sets the maximum number of descriptors in the cache; when the cache overflows, the least recently used (LRU) descriptors are closed.
`inactive`	Sets the time after which a cached descriptor is closed if it has not been accessed during this time. Default is 10 seconds.
`min_uses`	Sets the minimum number of file uses during the time specified by the `inactive` parameter, after which the file descriptor will remain open in the cache. Default is 1.
`valid`	Specifies after what time to check that the file still exists under the same name. Default is 60 seconds.
`off`	Disables caching.

Usage example:

```
open_log_file_cache max=1000 inactive=20s valid=1m min_uses=2;
```

Map

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.

Note

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 an unsuccessful attempt

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` *address*;
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*.

Note

If `memcached_pass` is placed in a `location` whose prefix ends with a slash (for example, `location /name/`), and the *auto_redirect* directive is set to `default`, requests without a trailing slash will be redirected (`/name -> /name/`).

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.

`off`	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 a response from a memcached server.

Metric

Added in version 1.11.0.

The `ngx_http_metric_module` module allows creating arbitrary real-time calculated metrics. These metric values are stored in shared memory and displayed in real-time within the `/status/http/metric_zones/` API branch. Various data aggregation types are supported (counters, histograms, moving averages, etc.) with grouping by arbitrary keys.

Configuration Example

Counting API requests:

```
http {
    metric_zone api_requests:1m count;

    server {
        listen 80;

        location /api/ {
            allow 127.0.0.1;
            deny all;
            api /status/;

            metric api_requests $http_user_agent on=request;
        }
    }
}
```

If a request is made to `/api/` with this configuration:

```
$ curl 127.0.0.1/api/ --user-agent "Firefox"
```

The `api_requests` metric is updated in real-time:

```
{
    "http": {
       "metric_zones": {
           "api_requests": {
               "discarded": 0,
               "metrics": {
                   "Firefox": 1
               }
           }
       }
    }
}
```

Directives

metric_zone

Syntax	`metric_zone` *name:size* [`expire=on| off`] [`discard_key=name`] *mode* [*parameters*];
Default	—
Context	http

Creates a shared memory zone of the specified *size* with the given *name* to store metrics. The zone name serves as a node in the `/status/http/metric_zones/` branch.

Parameters:

- `expire=<on|off>` — behavior when the zone is full:
  - **If `on`, the oldest metrics (by update time) are discarded to** free memory for new ones;
  - **If `off` (default) — new incoming metrics are discarded,** preserving existing entries.
- `discard_key=<name>` — **defines a metric with the key *name*** where values of discarded metrics are accumulated. By default, no such metric is created. Reserved key cannot be updated manually.
- *mode* — data processing algorithm (see the *Operation Modes* section);
- ***parameters* — additional settings for the selected mode** (e.g., `factor` for `average exp`).

Example usage:

```
metric_zone request_time:1m max;
```

In the API tree, the shared memory zone template looks as follows:

```
{
    "discarded": 0,
    "metrics": {
        "key1": 123,
        "key2": 10.5,
```

```
    }
}
```

`discarded`	Number; the count of discarded metrics in the shared memory zone
`metrics`	Object; its members are metrics with defined keys and calculated values

Note

In a 1 MB zone, with a key size of 39 bytes and a single metric mode, approximately 8,000 unique key entries can be stored.

metric_complex_zone

Syntax	`metric_complex_zone` *name:size* [`expire`=`on`| `off`] [`discard_key`=`name`] { ... }
Default	—
Context	http

Defines a *complex metric* — a set of metrics with independent modes. Each line in the block body defines a *submetric name*, a *mode*, and optional mode *parameters*.

Example usage:

```
metric_complex_zone requests:1m expire=on discard_key="old" {
    # submetric name   mode          parameters
    min_time           min;
    avg_time           average exp   factor=60;
    max_time           max;
    total              count;
}
```

In the API tree, such a complex metric template looks as follows:

```
{
    "discarded": 3,
    "metrics": {
        "key1": {
            "min_time": 20,
            "avg_time": 50,
            "max_time": 80,
            "total": 2
        },
        "old": {
             "min_time": 3,
             "avg_time": 40,
             "max_time": 152,
             "total": 80
        }
    }
}
```

`discarded`	Number; the count of discarded metrics in the shared memory zone
`metrics`	Object; its members are complex metrics with set keys. They are objects containing a set of submetrics with calculated values

metric

Syntax	`metric` *name key=value* [`on=request`| `response`| `end`];
Default	—
Context	http, server, location

Calculates the value of the metric for the specified shared memory zone *name*.

Parameters:

- ***key* — an arbitrary string (often a variable) used for grouping values.**
  Maximum length is 255 bytes. If the key is longer, it will be truncated to 255 bytes and appended with an ellipsis `...`;
- ***value* — a number (can be a variable) processed by the selected mode.**
  If omitted, it defaults to `0`. If the parameter cannot be converted to a number, it defaults to `1`;
- `on` — an optional parameter specifying when the metric is calculated:
  - If `on=request`, calculation occurs when the request is received;
  - If `on=response`, calculation occurs during response preparation;
  - If `on=end` (default), calculation occurs after the response is sent.

Note

In the case of internal redirection, metrics at the `on=request` stage are calculated in the original `location`. However, `on=response` and `on=end` metrics will be calculated in the new `location`.

Example usage:

```
metric requests $http_user_agent=$request_time;
```

Note

Metrics with an empty key or an invalid `key=value` pair are ignored. An omitted *value* is treated as `0`:

```
metric foo $bar;  # Equivalent to $bar=0
```

This is useful, for example, for the `count` mode, which ignores numerical values and simply reacts to the fact that a metric was updated.

Note

Remember that variables are evaluated in different phases. For example, it is impossible to use `$bytes_sent` (bytes sent to the client) with `on=request` (when the request is received).

Operation Modes

List of available metric operation modes:

- `count` — counter;
- `gauge` — gauge (increment/decrement);
- `last` — the last received value;

- `min` — minimum value;
- `max` — maximum value;
- `average exp` — exponential moving average (EMA) (parameter `factor`);
- `average mean` — mean over a window (parameters `window` and `count`);
- `histogram` — distribution across "buckets" (a list of threshold values).

count

The counter increases its value by `1` with every metric update.

Default value — `0`.

> **Note**
>
> Any metric update (with any value) monotonically increases the counter by `1`.

Examples:

```
metric_zone count:1m count;

# As part of a complex metric:
#
# metric_complex_zone count:1m {
#     some_metric_name  count;
# }

server {
    listen 80;

    location /metric/ {
        metric count KEY;
    }

    location ~ ^/metric/set/(.+)$ {
        metric count KEY=$1;
    }

    location /api/ {
        api /status/http/metric_zones/count/metrics/;
    }
}
```

Updating the metric:

```
$ curl 127.0.0.1/metric/
$ curl 127.0.0.1/metric/set/1
$ curl 127.0.0.1/metric/set/23
$ curl 127.0.0.1/metric/set/-32
```

Expected metric value in the API:

```
{
    "KEY": 4
}
```

gauge

The gauge increases or decreases its value depending on the sign of the passed number. A positive value increases the counter, while a negative value decreases it. A value of `0` does not change the counter.

Default value — `0`.

Examples:

```
metric_zone gauge:1m gauge;

# As part of a complex metric:
#
# metric_complex_zone gauge:1m {
#     some_metric_name  gauge;
# }

server {
    listen 80;

    location /metric/ {
        metric gauge KEY;
    }

    location ~ ^/metric/set/(.+)$ {
        metric gauge KEY=$1;
    }

    location /api/ {
        api /status/http/metric_zones/gauge/metrics/;
    }
}
```

Updating the metric:

```
$ curl 127.0.0.1/metric/
```

Expected metric value in the API:

```
{
    "KEY": 0
}
```

Further updates:

```
$ curl 127.0.0.1/metric/set/5
$ curl 127.0.0.1/metric/set/-5
$ curl 127.0.0.1/metric/set/8
```

Expected metric value in the API:

```
{
    "KEY": 8
}
```

last

Stores the last received value without any aggregation. If *value* is omitted, `0` is used.

Examples:

```
metric_zone last:1m last;

# As part of a complex metric:
#
# metric_complex_zone last:1m {
#     some_metric_name  last;
# }

server {
    listen 80;

    location /metric/ {
        metric last KEY;
    }

    location ~ ^/metric/set/(.+)$ {
        metric last KEY=$1;
    }

    location /api/ {
        api /status/http/metric_zones/last/metrics/;
    }
}
```

Updating the metric:

```
$ curl 127.0.0.1/metric/
```

Expected metric value in the API:

```
{
    "KEY": 0
}
```

Further updates:

```
$ curl 127.0.0.1/metric/set/8000
$ curl 127.0.0.1/metric/set/37
$ curl 127.0.0.1/metric/set/-3.5
```

Expected metric value in the API:

```
{
   "KEY": -3.5
}
```

min

Saves the minimum of two values — the currently stored value and the new one.

Examples:

```
metric_zone min:1m min;
```

```
# As part of a complex metric:
#
# metric_complex_zone min:1m {
#     some_metric_name  min;
# }

server {
    listen 80;

    location /metric/ {
        metric min KEY;
    }

    location ~ ^/metric/set/(.+)$ {
        metric min KEY=$1;
    }

    location /api/ {
        api /status/http/metric_zones/min/metrics/;
    }
}
```

Updating the metric:

```
$ curl 127.0.0.1/metric/set/42.999
$ curl 127.0.0.1/metric/set/-512
$ curl 127.0.0.1/metric/set/1
$ curl 127.0.0.1/metric/
```

Expected metric value in the API:

```
{
    "KEY": -512
}
```

max

Saves the maximum of two values — the currently stored value and the new one.

Examples:

```
metric_zone max:1m max;

# As part of a complex metric:
#
# metric_complex_zone max:1m {
#     some_metric_name  max;
# }

server {
    listen 80;

    location /metric/ {
        metric max KEY;
    }

    location ~ ^/metric/set/(.+)$ {
        metric max KEY=$1;
```

```
    }

    location /api/ {
        api /status/http/metric_zones/max/metrics/;
    }
}
```

Updating the metric:

```
$ curl 127.0.0.1/metric/set/42.999
$ curl 127.0.0.1/metric/set/-512
$ curl 127.0.0.1/metric/set/1
$ curl 127.0.0.1/metric/
```

Expected metric value in the API:

```
{
    "KEY": 42.999
}
```

average exp

Calculates the average value using the exponential smoothing algorithm.

Accepts an optional parameter `factor=<number>` — a coefficient determining how much the new value influences the average. Integer values from `0` to `99` are allowed. Default is `90`.

The higher the coefficient, the more weight new values have. If you specify `90`, the result will be `90%` of the new value and `10%` of the previous average.

Examples:

```
metric_zone avg_exp:1m average exp factor=60;

# As part of a complex metric:
#
# metric_complex_zone avg_exp:1m {
#     some_metric_name  average exp  factor=60;
# }

server {
    listen 80;

    location ~ ^/metric/set/(.+)$ {
        metric avg_exp KEY=$1;
    }

    location /api/ {
        api /status/http/metric_zones/avg_exp/metrics/;
    }
}
```

Updating the metric:

```
$ curl 127.0.0.1/metric/set/100
$ curl 127.0.0.1/metric/set/200
$ curl 127.0.0.1/metric/set/0
$ curl 127.0.0.1/metric/set/8
$ curl 127.0.0.1/metric/set/30
```

Expected metric value in the API:

```
{
    "KEY": 30.16
}
```

average mean

Calculates the arithmetic mean. Accepts optional parameters `window=<off|time>` and `count=<number>`, defining the time interval and sample size for averaging, respectively. Defaults: `window=off` (entire sample used) and `count=10`.

Note

For example, `window=5s` will only consider events from the last 5 seconds. The `window` parameter cannot be `0`. The `count=number` parameter controls the sample size (cached values) for a smoother mean calculation.

Examples:

```
metric_zone avg_mean:1m average mean window=5s count=8;

# As part of a complex metric:
#
# metric_complex_zone avg_mean:1m {
#     some_metric_name  average mean  window=5s count=8;
# }

server {
    listen 80;

    location ~ ^/metric/set/(.+)$ {
        metric avg_mean KEY=$1;
    }

    location /api/ {
        api /status/http/metric_zones/avg_mean/metrics/;
    }
}
```

Updating the metric:

```
$ curl 127.0.0.1/metric/set/0.1
$ curl 127.0.0.1/metric/set/0.1
$ curl 127.0.0.1/metric/set/0.4
$ curl 127.0.0.1/metric/set/10
$ curl 127.0.0.1/metric/set/1
$ curl 127.0.0.1/metric/set/1
```

Expected metric value in the API:

```
{
    "KEY": 2.1
}
```

If you wait 5 seconds from the last update, the expected value will be:

```
{
    "KEY": 0
}
```

histogram

Creates a set of "buckets," incrementing the relevant counter if the new value does not exceed the bucket threshold. Parameters are provided as a list of numerical thresholds. Useful for analyzing distributions, such as response times.

Mandatory parameters are *numbers* — the threshold values of the buckets, listed in ascending order.

Note

The bucket value `inf` or `+Inf` can be used to capture all values exceeding the highest specified bucket.

Examples:

```
metric_zone hist:1m histogram 0.1 0.2 0.5 1 2 inf;

# As part of a complex metric:
#
# metric_complex_zone hist:1m {
#     some_metric_name  histogram  0.1 0.2 0.5 1 2 inf;
# }

server {
    listen 80;

    location ~ ^/metric/set/(.+)$ {
        metric histogram KEY=$1;
    }

    location /api/ {
        api /status/http/metric_zones/hist/metrics/;
    }
}
```

Updating the metric:

```
$ curl 127.0.0.1/metric/set/0.25
```

Expected metric value in the API:

```
{
    "KEY": {
        "0.1": 0,
        "0.2": 0,
        "0.5": 1,
        "1": 1,
        "2": 1,
        "inf": 1
    }
}
```

Further updates:

```
$ curl 127.0.0.1/metric/set/2
```

Expected metric value in the API:

```
{
    "KEY": {
        "0.1": 0,
        "0.2": 0,
        "0.5": 1,
        "1": 1,
        "2": 2,
        "inf": 2
    }
}
```

Further update:

```
$ curl 127.0.0.1/metric/set/1000
```

Expected metric value in the API:

```
{
    "KEY": {
       "0.1": 0,
       "0.2": 0,
       "0.5": 1,
       "1": 1,
       "2": 2,
       "inf": 3
    }
}
```

Built-in Variables

Variables are created for each metric:

- `$metric_<name>`
- `$metric_<name>_key`
- `$metric_<name>_value`

For complex metrics, an additional variable is added:

- `$metric_<name>_value_<metric>`

`$metric_<name>`

Similar to the *metric* directive, the `$metric_<name>` variable setter can be used to update a metric. Calculation occurs during the Rewrite phase, allowing metric processing from the njs module, for example.

The value used for setting the variable must follow the `key=value` structure. Both key and value can consist of text, variables, and combinations thereof. The key is an arbitrary string for grouping values. The value is a number processed by the selected mode. If omitted, it defaults to `0`. If the parameter cannot be converted to a number, it defaults to `1`.

Example usage:

```
http {
    metric_zone counter:1m count;
```

```
    # At this point, the $metric_counter variable is added

    server {
        listen 80;

        location /metric/ {
            set $metric_counter $http_user_agent;  # Equivalent to $http_user_agent=0
        }

        location /api/ {
            allow 127.0.0.1;
            deny all;
            api /status/http/metric_zones/counter/;
        }
    }
}
```

Calculating metrics using the njs module:

```
http {
    js_import metrics.js;

    resolver 127.0.0.53;

    metric_complex_zone requests:1m {
        min_time        min;
        max_time        max;
        total           count;
    }

    location /metric/ {
        js_content metrics.js_request;
        js_fetch_trusted_certificate /path/to/ISRG_Root_X1.pem;
    }

    location /api/ {
        allow 127.0.0.1;
        deny all;
        api /static/http/metric_zones/requests/;
    }
}
```

File `metrics.js`:

```
async function js_request(r) {
    let start_time = Date.now();

    let results = await Promise.all([ngx.fetch('https://google.com/'),
                                     ngx.fetch('https://google.ru/')]);

    // Using the $metric_requests variable setter
    r.variables.metric_requests = `google={Date.now() - start_time}`;
}

export default {js_request};
```

After several requests to `location /metric/`, the values might look like this:

```
{
    "discarded": 0,
    "metrics": {
        "google": {
            "min_time": 70,
            "max_time": 432,
            "total": 6
        }
    }
}
```

Note

After setting the variable, you can retrieve its value; it will equal the specified `key=value` pair.

Additionally, the value stored in the `$metric_<name>_key` variable will change to the specified key.

`$metric_<name>_key` and `$metric_<name>_value`

The `$metric_<name>_key` and `$metric_<name>_value` variables define the key and value respectively. The metric update occurs when the `$metric_<name>_value` is set, provided that the key in `$metric_<name>_key` has already been defined.

Note

For complex metrics, the submetric values in the `$metric_<name>_value` variable are joined using a `", "` separator.

Example usage:

```
http {
    metric_zone gauge:1m gauge;

    # The variables $metric_gauge, $metric_gauge_key, and $metric_gauge_value are␣
↪added here.

    metric_complex_zone complex:1m {
        hist histogram 1 2 3;
        avg  average exp;
    }

    # $metric_complex, $metric_complex_key, and $metric_complex_value are added here.

    server {
        listen 80;

        location /gauge/ {
            set $metric_gauge_key "foo";
            set $metric_gauge_value 1;

            # Or: set $metric_gauge foo=1;

            return 200 "Updated with '$metric_gauge'\nValue='$metric_gauge_value'\n";
        }

        location /complex/ {
```

```
            set $metric_complex_key "foo";
            set $metric_complex_value 3;

            # Or: set $metric_complex foo=3;

            return 200 "Updated with '$metric_complex'\nValue='$metric_complex_value'\
↪n";
        }
    }
}
```

With this configuration, a request to /gauge/ yields:

```
$ curl 127.0.0.1/gauge/
Updated with 'foo=1'
Value='1'
```

For /complex/:

```
$ curl 127.0.0.1/complex/
Updated with 'foo=3'
Value='0 0 1, 3'
```

Note

If an empty string is assigned to $metric_<name>_value, the value is recognized as 0. If the string consists of characters that cannot be converted to a number, it is recognized as 1.

Calculation occurs only after both $metric_<name>_key and $metric_<name>_value have been set.

In this case, the value stored in $metric_<name> becomes equal to the new key=value pair.

The value in $metric_<name>_key represents the last key specified via variables.

The value in $metric_<name>_value represents the last calculated value for the key set in $metric_<name>_key.

$metric_<name>_value_<metric>

For complex metrics, the value of a specific submetric can be retrieved using the $metric_<name>_value_<metric> variable, where <*metric*> is the name of the submetric.

Example usage:

```
http {
    metric_complex_zone foo:1m {
        count count;
        min   min;
        avg   average exp;
    }

    # Adds $metric_foo, $metric_foo_key, $metric_foo_value,
    # and $metric_foo_value_count, $metric_foo_value_min, $metric_foo_value_avg.

    server {
        listen 80;

        location /foo/ {
            set $metric_foo_key   bar;
```

```
            set $metric_foo_value 9;

            # Or: set $metric_foo bar=9;

            return 200 "Updated with '$metric_foo'\nValues='$metric_foo_value'\nCount=
↪'$metric_foo_value_count'\n";
        }
    }
}
```

With this configuration, a request to `/foo/` yields:

```
$ curl 127.0.0.1/foo/
Updated with 'bar=9'
Values='1, 9, 9'
Count='1'
```

Additional Examples

Monitoring HTTP Methods

```
metric_zone http_methods:1m count;

server {
    listen 80;

    location / {
        metric http_methods $request_method;
    }

    location /metrics/ {
        allow 127.0.0.1;
        deny all;
        api /status/http/metric_zones/http_methods/metrics/;
    }
}
```

Response:

```
{
    "GET": 65,
    "POST": 20,
    "PUT": 10,
    "DELETE": 5
}
```

Upstream Response Time Distribution

```
metric_zone upstream_time:10m expire=on histogram
    0.05 0.1 0.3 0.5 1 2 5 10 inf;

server {
    listen 80;

    location /backend/ {
        proxy_pass http://backend;
        metric upstream_time $upstream_addr=$upstream_response_time on=end;
```

```
    }

    location /metrics/ {
        allow 127.0.0.1;
        deny all;
        api /status/http/metric_zones/upstream_time/;
    }
}
```

Response:

```
{
    "discarded": 0,
    "metrics": {
        "backend1:8080": {
            "0.05": 12,
            "0.1": 28,
            "0.3": 56,
            "0.5": 78,
            "1": 92,
            "2": 97,
            "5": 99,
            "10": 100,
            "inf": 100
        }
    }
}
```

Active Connections

```
metric_zone active_connections:2m gauge;

server {
    listen 80;
    server_name site1.com;

    location / {
        # Увеличиваем при подключении
        metric active_connections site1=1 on=request;

        # Уменьшаем при завершении
        metric active_connections site1=-1 on=end;
    }
}

server {
    listen 80;
    server_name site2.com;

    location / {
        metric active_connections site2=1 on=request;
        metric active_connections site2=-1 on=end;
    }
}

server {
    listen 8080;
```

```
    location /connections/ {
        allow 127.0.0.1;
        deny all;
        api /status/http/metric_zones/active_connections/metrics;
    }
}
```

Response:

```
{
    "site1": 42,
    "site2": 17
}
```

Prometheus Support

Angie includes a *built-in module* for displaying metrics in Prometheus format, which supports custom metrics.

As an integration example, consider the following configuration:

```
http {
    # Creating the "upload" metric
    metric_complex_zone upload:1m discard_key="other" {
        stats    histogram 64 256 1024 4096 16384 +Inf;
        sum      gauge;
        count    count;
        avg_size average exp;
    }

    # Describing the Prometheus template for the "upload" metric
    prometheus_template upload_metric {
        'stats{le="$1"}' $p8s_value
                         path=~^/http/metric_zones/upload/metrics/angie/stats/(.+)$
                         type=histogram;

        'stats_sum'      $p8s_value
                         path=/http/metric_zones/upload/metrics/angie/sum;
        'stats_count'    $p8s_value
                         path=/http/metric_zones/upload/metrics/angie/count;

        'avg_size'       $p8s_value
                         path=/http/metric_zones/upload/metrics/angie/avg_size;
    }

    server {
        listen 80;

        # Updating the metric
        location ~ ^/upload/(.*)$ {
            api /status/http/metric_zones/upload/metrics/angie/;
            metric upload angie=$1 on=request;
        }

        # Target for metric scraping
        location /prometheus/upload_metric/ {
            prometheus upload_metric;
```

```
        }
    }
}
```

After several requests to `/upload/...`:

```
$ curl 127.0.0.1/upload/16384
$ curl 127.0.0.1/upload/64448
$ curl 127.0.0.1/upload/64
$ curl 127.0.0.1/upload/1028
$ curl 127.0.0.1/upload/1028
```

The metric values will be:

```
{
    "stats": {
        "64": 1,
        "256": 1,
        "1024": 1,
        "4096": 3,
        "16384": 4,
        "+Inf": 5
    },

    "sum": 82952,
    "count": 5,
    "avg_size": 1077.9376
}
```

In Prometheus format, the metric is available at `/prometheus/upload_metric/`:

```
# Angie Prometheus template "upload_metric"
# TYPE stats histogram
stats{le="64"} 1
stats{le="256"} 1
stats{le="1024"} 1
stats{le="4096"} 3
stats{le="16384"} 4
stats{le="+Inf"} 5
stats_sum 82952
stats_count 5
avg_size 1077.9376
```

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.

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

Warning

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* module.

Configuration Example

```
location /video/ {
    mp4;
    mp4_buffer_size     1m;
    mp4_max_buffer_size 5m;
}
```

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 media data 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 allows writing location and variable handlers in Perl, as well as inserting 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`; it can be loaded using the *load_module* directive.

Note

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, so anything is possible.

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

Sets 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. The 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 Angie *statistics*, based on templates defined in the configuration, and returns metrics generated from these templates in the Prometheus format.

Warning

To collect statistics, enable a shared memory zone in the appropriate contexts using:

- the `zone` directive in *http_ upstream* or *stream_ upstream*;
- the *status_ zone* directive;
- the `status_zone` parameter in the *resolver* directive.

Configuration Example

Three metrics for collecting request statistics for server shared memory zones, combined into the `custom` template and published at the `/p8s` path:

```
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 includes a helper file `prometheus_all.conf` that contains a set of commonly used metrics combined into 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".';
}


'angie_http_acme_clients_state{client="$1"}' $p8st_acme_cert_state
    path=~^/http/acme_clients/([^/]+)/state$
    type=gauge
    'help=The current state of an ACME client: 1 - ready, 2 - requesting, 3 -␣
↪disabled, or 4 - failed.';

'angie_http_acme_certs_state{client="$1"}' $p8st_acme_cli_state
    path=~^/http/acme_clients/([^/]+)/certificate$
    type=gauge
    'help=The current state of an ACME client certificate: 1 - valid, 2 - mismatch, 3␣
↪- expired, 4 - missing, or 5 - error.';


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_acme_cli_state {
    volatile;
    "ready"        1;
    "requesting"   2;
    "disabled"     3;
    "failed"       4;
}


map $p8s_value $p8st_acme_cert_state {
    volatile;
    "valid"        1;
    "mismatch"     2;
    "expired"      3;
    "missing"      4;
    "error"        5;
}
```

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.';


'angie_stream_upstreams_backup_switch_active{upstream="$1"}' $p8s_value
    path=~^/stream/upstreams/([^/]+)/backup_switch/active$
    type=gauge
    'help=The currently active "stream" upstream servers backup group level.';
}


'angie_http_acme_clients_state{client="$1"}' $p8st_acme_cert_state
    path=~^/http/acme_clients/([^/]+)/state$
    type=gauge
    'help=The current state of an ACME client: 1 - ready, 2 - requesting, 3 -␣
↪disabled, or 4 - failed.';

'angie_http_acme_certs_state{client="$1"}' $p8st_acme_cli_state
    path=~^/http/acme_clients/([^/]+)/certificate$
```

```
    type=gauge
    'help=The current state of an ACME client certificate: 1 - valid, 2 - mismatch, 3␣
↪- expired, 4 - missing, or 5 - error.';


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_acme_cli_state {
    volatile;
    "ready"        1;
    "requesting"   2;
    "disabled"     3;
    "failed"       4;
}


map $p8s_value $p8st_acme_cert_state {
    volatile;
    "valid"        1;
    "mismatch"     2;
    "expired"      3;
    "missing"      4;
    "error"        5;
}


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_name*;
Default	—
Context	location

Specifies a template handler for the `location` context, defined by the *prometheus_template* directive. When requested, this `location` calculates and returns the template metrics in Prometheus format.

```
location =/p8s {
    prometheus custom;
}
```

```
$ curl localhost/p8s

    # Angie Prometheus template "custom"
    ...
```

prometheus_template

Syntax	`prometheus_template` *template_name* { ... }
Default	—
Context	http

Defines a named template of metrics collected and exported by Angie, for use with the *prometheus* directive.

Note

Angie also includes a ready-made *all* template that contains a set of the most commonly used metrics.

Can contain any number of metric definitions, each having the following structure: <*metric_name*> <*variable*> [`path=`<*match_string*>] [`type=`<*type*>] [`help=`<*help*>].

`metric_name`	Sets the metric name under which it will be added in Prometheus format to the response. Can contain an optional labels section (`...`), for example: `http_requests_total{method="$1",code="$2"}` Label values can use Angie variables; if *match_string* is defined as a regular expression, you can also use capture groups defined in that expression. Such variables and groups are evaluated when obtaining the metric value, which is set by *variable*.
`variable`	Sets the name of the variable that will be evaluated and added as the metric value to the response. If the variable doesn't exist or the evaluation result is empty (`""`), the metric is not added.

The metric is calculated with the value set by *variable*; upon successful evaluation, the metric is added to the response, for example:

```
'angie_time{version="$angie_version"}' $msec;
```

```
$ curl localhost/p8s

   angie_time{version="1.11.8"} 1695119820.562
```

`path=match_strin`	Is matched against all endpoint paths of metrics in the */status* API subtree of Angie, allowing multiple instances of the metric to be added to the response at once.

During matching, paths are taken with the leading slash but without the trailing one, for example `/angie/generation`; matching is case-insensitive. There are two matching methods:

`path=exact_match`	Checked by character-by-character comparison.
`path=~regular_ex`	Checked using the PCRE library; can define capture groups for use in the labels of the *metric_name* field.

If *match_string* matches any path, the value of the Angie metric at that path is stored in the *$p8s_value* variable, which can be used in the *variable* field when `path=` is specified.

If *match_string* ends with a trailing slash, the metric value is the number of items in the corresponding list or object. For example:

```
'angie_http_server_zones_count' $p8s_value
    path=/http/server_zones/;
```

In the case of regular expressions, there can be multiple matching paths; the metric is added to the response for *each* match. Combined with capture groups, this allows obtaining a series of metrics with the same name and different labels, for example:

```
'angie_slabs_slots_free{zone="$1",size="$2"}' $p8s_value
    path=~^/slabs/([^/]+)/slots/([^/]+)/free$;
```

This definition adds metrics for all zones and all sizes that currently exist in the configuration:

```
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 (with any matching method), the metric is not added.

Note

The `path=` parameter is available only when Angie is built with the *API* module.

`type=type`, `help=help`	Set the metric's type and help string, respectively, in the Prometheus format, which are added with the metric to the response without changes or validation.

Built-in Variables

The `http_prometheus` module has a built-in variable that receives its value when matching metric paths from the */status* section of the Angie API with the *match_string* parameter of metrics defined by the *prometheus_template* directive.

`$p8s_value`

If the *match_string* of a metric defined in *prometheus_template* matches any path, the value of the Angie metric located at that path is stored in the `$p8s_value` variable. It is intended for use in the *variable* field in metric definitions that are calculated based on the `path=` parameter.

The values of Angie metrics stored in the `$p8s_value` variable do not always meet the requirements of the Prometheus format. In such cases, you can use the *map* directive, for example to convert strings to 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 has a boolean value, that is `true` or `false`, the variable receives the value `"1"` or `"0"` respectively; if the metric value is `null`, the variable will be `"(null)"`. For dates, the integer UNIX 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;
```

```
    proxy_cache       cache_zone;
    proxy_cache_valid 200 302 10m;
    proxy_cache_valid 404      1m;
}
```

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.

Note

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. Sending to the client is performed in parallel: filled buffers are passed for sending, but their total size is limited by *proxy_busy_buffers_size*. If a buffer is not completely filled, it is not passed for sending unless it contains the last part of the response. Therefore, buffered reading is not suitable when you need immediate delivery of every few bytes. 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 immediately as it is received. Angie works in a "read — send" loop and does not wait for the buffer to fill completely: for example, 10 bytes read from a 4K buffer are sent right away. At the same time, if the entire response fits into the buffer, Angie can read it in full. The maximum size of the data that Angie can receive from the server at a time is set by the *proxy_buffer_size* directive. With `off`, Angie does not cache responses and *proxy_limit_rate* does not work.

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 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.

In Angie PRO, you can specify multiple *proxy_cache_path* directives that use the same `keys_zone` value to enable *cache sharding*. When doing so, you should set the `path` parameter of the *proxy_cache* directive that uses this `keys_zone` value:

`path=`*path*	The value is evaluated at the time of *caching* the response from the backend and is expected to use variables, including those containing information from the response. If the response is taken from the cache, the *path* is not reevaluated; thus, a cached response retains its original *path* until it is removed from the cache.

This allows selecting the needed cache path by applying *map* directives or scripts to responses from the backend. Example for `Content-Type`:

```
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;
       proxy_cache_valid 200 10m;
   }
}
```

Here there are two cache paths and a variable mapping to distinguish between them. If `Content-Type` starts with `text/`, the first path will be chosen, otherwise the second.

Note

When using *proxy_cache*, you typically need to also set the *proxy_cache_valid* directive to explicitly specify the caching time for responses. If it's not set, Angie doesn't use default values but determines the response caching time based on HTTP response headers from the server in the following priority order:

1. The `X-Accel-Expires` header (highest priority).
2. The `Cache-Control` header with `max-age` or `s-maxage` parameters.
3. The `Expires` header.

If none of these headers contain valid values or are not present at all, the response will not be cached because its expiration time cannot be determined.

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.

Warning

The use of a stale cached response while it's being updated must be *allowed*.

proxy_cache_bypass

Syntax	`proxy_cache_bypass ...;`
Default	—
Context	http, server, location

Defines conditions under which the response will not be taken from 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 cache:

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

Can be used together 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. If conversion is disabled, the *cache key* must 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 the proxied server. Other requests for the same cache element will either wait for a response to appear in cache or for 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 specified 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, but 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.

Warning

Cache metadata is stored in shared memory. Manually deleting cache files does not reset the counters and may lead to unpredictable behavior. For a complete reset, stop the server, delete the cache directory, and start again.

Note

Third-party cache purge modules (for example, Cache Purge) only delete files but do not reset the *proxy_cache_min_uses* counter. The directive is intended to protect the cache from pollution by infrequent requests, and resetting the counter on purge may negatively affect performance.

proxy_cache_path

Changed in version 1.9.0.

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 | off`	determines the directory to be used for temporary files
`on`	If the parameter is not specified or set to `on`, the directory specified by the *proxy_temp_path* directive for the given *location* will be used
`off`	temporary files will be placed directly in the cache directory
`keys_zone`	Sets the name and size of the shared memory zone for storing all active keys and information about data. Cache metadata is stored in shared memory. A zone of 1 megabyte is sufficient to store about 8000 keys. When an optional *file* is specified with `keys_zone`, Angie dumps the contents of this zone to disk when the main process terminates and attempts to restore it at the same memory address on the next *startup* or after *binary upgrade*, to ensure more reliable data persistence and reduce cache loading time. If the zone cannot be restored due to a change in its size, binary version incompatibility, or other reasons, Angie will log a warning (`failed to restore zone at address`) and will not use the zone restoration mechanism. Instead, the incompatible file will be renamed to `.old`; you can either delete it, or restore its name and revert Angie to the configuration and version in which this file was originally created. **Warning** Make sure the path to the *file* is specified correctly and has the necessary access permissions for Angie to use, and is protected from unauthorized access; relative paths are based on the prefix.
`inactive`	If cached data is not accessed within the time specified by this parameter, the data is removed, regardless of its freshness. By default, `inactive` is 10 minutes.

Note

In Angie PRO, multiple *proxy_cache_path* directives can be specified with the same `keys_zone` value. The shared memory zone size can only be specified in the first one. Selection between directives will be made based on the `path` parameter of the corresponding *proxy_cache* directive.

A special **cache manager** process monitors the maximum cache size and the minimum amount of free space on the filesystem with the cache, and removes the least recently used data when the maximum cache size is exceeded or there is insufficient free space. Data removal occurs in iterations.

`max_size`	maximum threshold value for cache size
`min_free`	minimum threshold value for free space on the filesystem with the cache
`manager_files`	maximum number of cache items to be removed in 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.

One minute after Angie starts, a special **cache loader** process is activated. It scans the filesystem for previously cached data and loads this information into the cache zone. This process works iteratively; each iteration processes a limited number of items specified by the `loader_files` parameter, ensures it doesn't exceed `loader_threshold`, then takes a short pause defined by `loader_sleep` before moving to the next batch. Iterations continue until the loader has processed all existing cache entries on disk:

`loader_files`	maximum number of cache items to load in 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

Specifying the *file* for the `keys_zone` parameter does not affect the cache loader operation.

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. The directive parameters match those 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 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".

The leading dot in the *domain* and *replacement* strings, as well as in the *domain* attribute, is ignored. The value 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 the "~" symbol. The 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 one 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 on | 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.

Warning

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 directive *http3_max_table_capacity* sets this value for server connections. To avoid errors, dynamic table usage is disabled when caching is enabled in proxy mode.

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 buffer used for reading and writing *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; variables are allowed.

`0`	disables rate limiting

Note

The limit is set per a request, 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

Changed in version 1.11.0: If all servers in the upstream group are unavailable or respond with a status considered an error by this directive (and its counterparts for other protocols), Angie now always returns its own error page instead of the response from the last server.

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* group:

`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.

If all upstream servers are unavailable or return a response with a status considered an error by `proxy_next_upstream`, Angie returns its own standard error page instead of the response body from the last upstream server.

Note

It should be understood 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`
	are always considered unsuccessful attempts, even if they are not specified in the directive

`http_500`
`http_502`
`http_503`
`http_504`
`http_429`
	are considered unsuccessful attempts only if they are specified in the directive

`http_403` `http_404`	are never considered unsuccessful attempts

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`	disables 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`	disables 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 `location`.

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.

Note

If `proxy_pass` is placed in a `location` with a trailing slash in the prefix (for example, `location /name/`), and the *auto_redirect* directive is set to `default`, requests without a trailing slash will be redirected (`/name -> /name/`).

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/;
}
```

Warning

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.

`off`	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.

When *proxy_ssl_ntls* is enabled, the directive accepts two arguments instead of one:

```
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.

The value `store:scheme:id` can be specified instead of the file, which is used to load a secret key with a specified id and OpenSSL provider registered URI scheme, such as pkcs11.

Variables can be used in the file name.

When *proxy_ssl_ntls* is enabled, the directive accepts two arguments instead of one:

```
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.

Warning

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.

Note

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.

Warning

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

Syntax	`proxy_ssl_ntls on | off;`
Default	`proxy_ssl_ntls off;`
Context	http, server

Enables client-side support for NTLS using the TongSuo TLS 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;
}
```

Note

Angie must be built with the `--with-ntls` configuration parameter and the corresponding SSL library with NTLS support

```
./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

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) when 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 certificate 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 according to the paths specified in the *alias* or *root* directives
`off`	disables saving of files

The file name can be set explicitly using a `string` with variables:

```
proxy_store /data/www$original_uri;
```

The modification time of files is set according to the received `Last-Modified` header field in the response. The response is first written to a temporary file, and then the file is renamed. The temporary file and the persistent store for the response can be on different file systems. However, be aware that in this case instead of the cheap rename operation within one file system, the file is copied from one file system to another. 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 build option `--http-proxy-temp-path`)
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* module 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.

Warning

Although directives from other modules can be used inside the *if* block, this is not recommended, as it may lead to unexpected behavior.

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

The 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* module directives and starts a search for a new *location* matching the changed URI;
`break`	stops processing the current set of *http_rewrite* module 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 for zero
    return 403
    end of code
variable $slow
check for 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, since it is not related to the `http_rewrite` module. A separate configuration is created for the `if` block. If the condition is true, the request gets this configuration, where `limit_rate` equals 10k.

The directive

```
rewrite ^/(download/.*)/media/(.*)\..*$ /$1/mp3/$2.mp3 break;
```

can be made one instruction shorter if the first slash is moved inside the parentheses in the regular expression:

```
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 an 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 an 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 an 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.

Note

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 on` | `off`;
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. Sending to the client is performed in parallel: filled buffers are passed for sending, but their total size is limited by *scgi_busy_buffers_size*. If a buffer is not completely filled, it is not passed for sending unless it contains the last part of the response. Therefore, buffered reading is not suitable when you need immediate delivery of every few bytes. 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 immediately as it is received. Angie works in a "read — send" loop and does not wait for the buffer to fill completely: for example, 10 bytes read from a 4K buffer are sent right away. At the same time, if the entire response fits into the buffer, Angie can read it in full. The maximum size of the data that Angie can receive from the server at a time is set by the *scgi_buffer_size* directive. With `off`, *scgi_limit_rate* does not work.

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.

Warning

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` *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:

```
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 an 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.

Warning

Cache metadata is stored in shared memory. Manually deleting cache files does not reset the counters and may lead to unpredictable behavior. To completely reset the cache, stop the server, delete the cache directory, and restart.

Note

Third-party cache purge modules (e.g., Cache Purge) only delete files but do not reset the *scgi_cache_min_uses* counter. The directive is intended to protect the cache from pollution by infrequent requests, and resetting the counter during purge may negatively affect performance.

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. Cache metadata is stored in shared memory.

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_503 | 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. 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 method has lower priority than setting parameters using the directive.

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, it can be specified to cache any responses using the `any` parameter:

```
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 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 an 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 on` | `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 an SCGI server to a client. The `scgi_hide_header` directive sets additional fields that will not be passed. If, conversely, 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 an 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 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.

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; variables can be used.

`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:

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

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

`http_403` `http_404`	are 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;
#    ...
}
```

In the standard `scgi_params` file, `REQUEST_METHOD` is set to `$upstream_request_method`.

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

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:

```
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*.

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*.

Note

If `scgi_pass` is specified in a `location` with a trailing slash in the prefix (for example, `location /name/`), and the *auto_redirect* directive is set to `default`, requests without a trailing slash will be redirected (`/name -> /name/`).

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 request body is fully *read* from the client before sending the request to an 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 has 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.

`off`	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 allows checking authenticity of requested links, protecting resources from unauthorized access, and limiting 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 implements two alternative operation modes. The first mode is enabled by the *secure_link_secret* directive and allows checking authenticity of requested links and protecting them from unauthorized access. The second mode is enabled by the *secure_link* and *secure_link_md5* directives and also allows limiting link lifetime.

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 do not match, the *$secure_link* variable is set to an empty string. If the checksums match, 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 (January 1, 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 use in the *secure_link_md5* directive. If the expiration time is not specified, a link has 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 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 large 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 generates variables for A/B testing, canary releases, and other scenarios that direct a certain percentage of clients to one server or configuration while routing 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 *$variable* by hashing the *string*; variables in the *string* are substituted, the result is hashed, then the hash value is used to select the string value of the *$variable*.

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 don't fall into other buckets are mapped to its assigned value.

Example:

```
split_clients "${remote_addr}AAA" $variant {
              0.5%               .one;
              2.0%               .two;
              *                  "";
}
```

Here, after substitution in the `$remote_addrAAA` string, the hash values are distributed as follows:

- values from 0 to 21474835 (0.5%) yield `.one`;
- values from 21474836 to 107374180 (2%) yield `.two`;
- values from 107374181 to 4294967295 (all others) yield `""` (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.

Note

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.

Note

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*.

Note

It should be kept in mind that due to the HTTPS protocol limitations for maximum interoperability virtual servers should listen on *different IP addresses*.

If *ssl_ ntls* is enabled, the directive can accept two arguments (the signature and the encryption parts of the certificate) 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_key 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_compression

Syntax	`ssl_certificate_compression on	off;`
Default	`ssl_certificate_compression off;`
Context	http, server

Enables TLS 1.3 compression of server certificates.

Note

The directive is supported when using OpenSSL 3.2 or higher; the list of supported compression algorithms is provided by the library.

Note

The directive is supported when using BoringSSL; the list of supported compression algorithms includes `zlib`.

If *ssl_stapling* is enabled, certificate compression is disabled.

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.

Note

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 `store:scheme:id` can be specified instead of the *file*, which is used to load a secret key with a specified *id* and OpenSSL provider registered URI *scheme*, such as pkcs11.

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*.

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_key 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.

Warning

The `ssl_ciphers` directive does *not* configure ciphers for TLS 1.3 when using OpenSSL. To configure 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.

Note

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.

Warning

Note that 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.

Warning

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;
```

Note

The directive is supported when using OpenSSL 1.1.1 or higher or BoringSSL.

ssl_encrypted_hello_key

Added in version 1.11.0.

Syntax	`ssl_encrypted_hello_key` *file*;
Default	—
Context	http, server

Specifies a file with an ECH private key and ECHConfigList in PEM format. The directive can be specified multiple times. Requires an OpenSSL or BoringSSL build with Encrypted Client Hello (ECH) support; otherwise it is not supported.

ssl_ecdh_curve

Syntax	`ssl_ecdh_curve` *curve*;
Default	`ssl_ecdh_curve auto;`
Context	http, server

Specifies a curve for ECDHE ciphers.

Note

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` corresponds to the list of curves built into the OpenSSL library for OpenSSL 1.0.2 or higher, or *prime256v1* for older versions.

Note

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

Syntax	`ssl_ntls on` | `off`;
Default	`ssl_ntls off;`
Context	http, server

Enables server-side support for NTLS when using the TongSuo TLS library.

```
listen ... ssl;
ssl_ntls  on;
```

Note

Angie must be built with the `--with-ntls` configuration parameter, with the corresponding SSL library with NTLS support

```
./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 the name and size of the cache that stores client certificate 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

Enables the specified protocols.

Note

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 needs 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) will be 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 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 file specified by the *ssl_trusted_certificate* directive.

Warning

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 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

Note

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_sigalg`

returns the signature algorithm for the client certificate for an established SSL connection.

Note

The variable is supported only when using OpenSSL version 3.5 or higher. With older versions, the variable value will be an empty string.

Note

The variable is available only for new sessions.

`$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 key exchange during the SSL handshake. Known curves are listed by names, unknown are shown in hexadecimal, for example:

prime256v1

Note

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

Note

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_encrypted_hello`

Added in version 1.11.0.

returns "1" if Encrypted Client Hello (ECH) is used, 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_sigalg`

returns the signature algorithm for the server certificate for an established SSL connection.

Note

The variable is supported only when using OpenSSL version 3.5 or higher. With older versions, the variable value will be an empty string.

Note

The variable is available only for new sessions.

Stub Status

The module provides access to basic server 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 information which may look 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 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

Provides a context for describing groups of servers that can be used in 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 the ability to start server selection not from the primary group, but from the *active* group, i.e., the one where a server was successfully found last time. If a server cannot be found in the active group for the next request, and the search moves to the backup group, then this group becomes active, and subsequent requests are first directed to servers in this group.

If the `permanent` parameter is defined without a *time* value, the group remains active after selection, and automatic rechecking of groups with lower levels does not occur. If *time* is specified, the active status of the group expires after the specified interval, and the balancer again checks groups with lower levels, returning to them if the servers are working normally.

Example:

```
upstream my_backend {
    zone my_backend 1m;
    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 switches from primary servers to the backup group, all subsequent requests are handled by this backup group for 2 minutes. After 2 minutes elapse, the balancer rechecks the primary servers and makes them active again if they are working normally.

bind_conn (PRO)

Syntax	`bind_conn` *value*;
Default	—
Context	upstream

Allows binding a server connection to a client connection when the *value*, specified as a string of variables, becomes different from `""` and `"0"`.

Warning

The `bind_conn` directive must be used after all directives that set a load balancing method, otherwise it will not work. If it is used together with the *sticky* directive, then `bind_conn` must come after `sticky`.

Warning

When using the directive, the *Proxy* module settings must allow the use of persistent connections, for example:

```
proxy_http_version 1.1;
proxy_set_header Connection "";
```

A typical use case for the directive is proxying connections with NTLM authentication, where it is necessary to ensure client-to-server binding at the beginning of negotiation:

```
map $http_authorization   $ntlm {
    ~*^N(?:TLM|egotiate)  1;
}

upstream ntlm_backend {
    zone ntlm_backend 1m;
    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)

Syntax	`feedback` *variable* [`inverse`] [`factor=`*number*] [`account=`*condition_variable*] [`last_byte`];
Default	—
Context	upstream

Sets up a feedback-based load balancing mechanism in the `upstream`. It dynamically adjusts balancing decisions by multiplying the weight of each proxied server by the average feedback value, which changes over time depending on the value of the *variable* and is subject to an optional condition.

The following parameters can be specified:

`variable`	The variable from which the feedback value is taken. It should represent a performance or health metric; it is assumed that the server provides it in headers or otherwise. The value is evaluated with each response from the server and is factored into the moving average according to the `inverse` and `factor` settings.
`inverse`	If the parameter is set, the feedback value is interpreted inversely: lower values indicate better performance.
`factor`	The factor by which the feedback value is considered when calculating the average. Valid values are integers from 0 to 99. Default is `90`. The average is calculated using the exponential smoothing formula. The larger the factor, the less new values affect the average; if `90` is specified, then 90% of the previous value and only 10% of the new value will be taken.
`account`	Specifies a condition variable that controls which responses are considered in the calculation. The average value is updated with the feedback value from the response only if the condition variable of that response is not equal to `""` or `"0"`. **Note** By default, responses during *active checks* are not included in the calculation; combining the *$upstream_probe* variable with `account` allows including these responses or even excluding everything else.
`last_byte`	Allows processing data from the proxied server after receiving the complete response, not just 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 configuration categorizes server responses by feedback levels based on specific scores from response header fields, and also adds a condition on *$upstream_probe* to consider only responses from the `high_priority` active check 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 determined using 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.

```
hash $remote_addr;
```

When using domain names that resolve to multiple IP addresses (for example, with the `resolve` parameter), the server does not sort the received addresses, so their order may differ across different servers, which affects client distribution. To ensure consistent distribution, use the `consistent` parameter.

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 a load balancing method for the group where requests are distributed among servers based on client IP addresses. The first three octets of the client's 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 that case, client requests will be passed to another server. Most likely this will also be the same server.

If one of the servers needs to be temporarily removed, it should be marked with the `down` parameter to preserve the current hashing of client IP addresses:

```
upstream backend {
    zone backend 1m;
    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 of 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 Angie worker processes can open. The `connections` parameter should be set low enough to let upstream servers process new incoming connections as well.

Warning

The `keepalive` directive must be used after all directives that set a load balancing method, otherwise it will not work.

Example configuration of memcached upstream with keepalive connections:

```
upstream memcached_backend {
    zone memcached_backend 1m;
    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 {
    zone http_backend 1m;
    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 {
    zone fastcgi_backend 1m;
    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 concept 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` *timeout*;
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 the average time to receive response headers.
`last_byte`	The directive uses the average time to receive the entire response.

`factor`	Serves the same purpose as *response_time_factor (PRO)* and overrides it if the parameter is set.
`account`	Specifies a condition variable that controls which responses are included in the calculation. The average is updated only if the condition variable for the response is not `""` or `"0"`.

Note

By default, responses during *active health probes* are not included in the calculation; combining the *$upstream_probe* variable with `account` allows including these responses or even excluding everything else.

Current values 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.

queue (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 parameter of 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 the states of requests passing through the queue in the *API*.

Warning

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 a load balancing method for the group where a request is passed to a randomly selected server, taking into account server weights.

If the optional `two` parameter is specified, Angie randomly selects two servers, then selects one of them using the *least_conn* method, where a request is passed to the 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 **previous** value when calculating the average response time for the *least_time (PRO)* load balancing method using the exponentially weighted moving average formula.

The higher the specified *number*, the less new values affect the average; if `90` is specified, 90% of the previous value and only 10% of the new value are taken. Valid values range from 0 to 99 inclusive.

Current calculation results 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

Only successful responses are included in the calculation; what is considered an unsuccessful response is determined by the *proxy_next_upstream*, *fastcgi_next_upstream*, *uwsgi_next_upstream*, *scgi_next_upstream*, *memcached_next_upstream*, and *grpc_next_upstream* directives. Additionally, the `header_time` value is recalculated only if all headers are received and processed, and `response_time` — only if the entire response 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, port 80 is used. A domain name that resolves to multiple IP addresses defines multiple servers at once.

The following parameters can be defined:

`weight=`*number*	Sets the weight of the server. Default is 1.
`max_conns=`*number*	Limits the maximum number of simultaneous active connections to the proxied server. The default value is `0`, meaning there is no limit. If the group does not reside in the *shared memory zone*, the limitation works per worker process.

Note

With *idle keepalive connections* enabled, multiple *worker processes*, and a *shared memory zone*, 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 occur during the specified *fail_timeout* period for the server to be considered unavailable; after this, it will be checked again after the same period.

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 exceeded, the server is also considered unavailable from the perspective of *upstream_probe (PRO)*; client requests will not be directed to it until checks determine it is available.

Note

If a `server` directive in a group resolves to multiple servers, its `max_fails` setting applies to each server separately.

If after resolving all `server` directives only one server remains in the upstream, the `max_fails` setting has no effect and will be ignored.

`max_fails=1`	Default number of attempts;
`max_fails=0`	Disables attempt accounting.

`fail_timeout=`*time* — sets the period of time during which a specified number of unsuccessful attempts to communicate with the server (*max_fails*) must occur for the server to be considered unavailable. The server then remains unavailable for the same period of time before being checked again.

Default value is 10 seconds.

Note

If a `server` directive in a group resolves to multiple servers, its `fail_timeout` setting applies to each server separately.

If after resolving all `server` directives only one server remains in the upstream, 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 the *backup_switch (PRO)* directive is specified, its active backup logic also applies.
`down`	Marks the server as permanently unavailable.
`drain` (PRO)	Marks the server as draining; this means it only receives requests from sessions previously bound via *sticky*. Otherwise, the behavior is the same as in `down` mode.

Warning

The `backup` parameter cannot be used together with the *hash*, *ip_hash*, and *random* load balancing methods.

The `down` and `drain` parameters are mutually exclusive.

`resolve`	Allows monitoring changes to the list of IP addresses corresponding to a domain name and updating it without reloading the configuration. The group must reside in *shared memory*; a *resolver* must also be defined.
`service=`*name*	Enables resolution of DNS SRV records and sets the service name. For the parameter to work, the `resolve` parameter must be specified for the server, without specifying a port in the hostname. If the service name does not contain a dot, a name is formed according to the RFC standard: the service name is prefixed with `_`, then `_tcp` is appended after a dot. Thus, the service name `http` results in `_http._tcp`. Angie resolves SRV records by combining the normalized service name and hostname and obtaining a list of servers for the resulting combination via DNS, along with their priorities and weights. • SRV records with the highest priority (those with the lowest priority value) are resolved as primary servers, while other records become backup servers. If `backup` is set with `server`, SRV records with the highest priority are resolved as backup servers, and other records are ignored. • Weight is analogous to the `weight` parameter of the `server` directive. If weight is specified both in the directive itself and in the SRV record, the weight set in the directive is used.

In this example, a lookup is performed for the record `_http._tcp.backend.example.com`:

```
server backend.example.com service=http resolve;
```

`sid=`*id*	Sets the server ID in the group. If the parameter is not specified, the ID is set as a hexadecimal MD5 hash of the IP address and port or UNIX socket path.

`slow_start=`*time*	Sets the *time* for gradual recovery of a returning server's weight when load balancing with the *round-robin* or *least_conn* method. If the parameter is set and the server is considered operational again after a failure from the perspective of *max_fails* and *upstream_probe (PRO)*, such a server gradually increases to its specified weight over the given time period. If the parameter is not set, in a similar situation the server immediately starts operating with its specified weight.

Note

If only one `server` is specified in the upstream, `slow_start` does not work and will be ignored.

state (PRO)

Syntax	`state` *file*;
Default	—
Context	upstream

Specifies a *file* where the list of upstream servers is persistently stored. When installing from our packages, the directory `/var/lib/angie/state/` (`/var/db/angie/state/` on FreeBSD) is created specifically for storing such files with appropriate access permissions, and in the configuration you only need to add the file name:

```
upstream backend {

    zone backend 1m;
    state /var/lib/angie/state/<FILE NAME>;
}
```

The server list here has a format similar to `server`. The file contents are modified whenever servers are changed in the */config/http/upstreams/* section via the configuration API. The file is read when Angie starts or when the configuration is reloaded.

Warning

To use the `state` directive in an `upstream` block, there must be no `server` directives in it, but a shared memory zone (*zone*) is required.

sticky

Changed in version 1.10.0: PRO

Changed in version 1.11.0.

Changed in version 1.11.0: PRO

Syntax	`sticky` cookie *name* [*attr=value*]...; `sticky route` *value*...; `sticky learn zone=`*zone* `create=`*$create_var1*... `lookup=`*$lookup_var1*... [`header`] [`norefresh`] [`timeout=`*time*]; `sticky learn` [`zone=`*zone*] `lookup=`*$lookup_var1*... `remote_action=`*uri* `remote_result=`*$remote_var* [`norefresh`] [`timeout=`*time*];
Default	—
Context	upstream

Configures session affinity to bind client sessions to proxied servers in the mode specified by the first parameter; to drain servers that have the `sticky` directive configured, you can use the `drain` (PRO) option in the *server* block.

Warning

The `sticky` directive must be used after all directives that specify a particular load balancing method, otherwise it will not work. If it is used alongside the *bind_conn (PRO)* directive, then `bind_conn` must come after `sticky`.

`cookie` mode

This mode uses cookies to manage sessions. It's suitable when cookies are already used for session tracking.

The first client request, before any stickiness applies, is sent to a backend server according to the configured balancing method. Angie then sets a cookie identifying the chosen server.

The cookie name (`name`) is defined by the `sticky` directive, and its value corresponds to the *sid* from the *server* directive. This value is further hashed if *sticky_secret* is set.

Subsequent requests with this cookie are routed to the server specified by its *sid*. If the server is unavailable or can't process the request, another one is chosen via the configured balancing method.

You can assign cookie attributes in the directive; by default, only `path=/` is set. Attribute values can contain variables. To remove an attribute, set it to an empty value: `attr=`. For example, `sticky cookie path=` omits the `path` attribute.

This example sets a cookie `srv_id` for 1 hour, using a domain from a variable:

```
upstream backend {
    zone backend 1m;
    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, which may come from URLs, cookies, or request arguments. It's less flexible but good when such identifiers already exist.

The backend server may return a route ID known to both client and server. This value must match the *sid*.

Subsequent requests should carry the route ID, e.g., via a *cookie* or *query argument*.

The directive takes a list of strings that may include variables to extract the route ID. The first non-empty result is matched against *sid*.

In this example, Angie checks the `route` cookie first, then the `route` query argument:

```
upstream backend {
    zone backend 1m;
    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 assign a client to a backend. It's flexible and supports session storage in shared memory and various identifier sources.

A session is created from the backend server's response. `create` and `lookup` define how to generate and locate sessions, and both accept multiple variables.

The session ID is the first non-empty variable from `create`. For example, it may come from a response cookie.

Sessions are stored in shared memory, defined with `zone name:size`. If unused for `timeout` duration (default: 1 hour), the session expires.

By default, Angie refreshes sessions on each use. To disable this, use `norefresh`.

The session ID from a client request is extracted via `lookup`, using the first non-empty variable listed. If none is found, it's a new request.

Use `header` to create the session upon receiving response headers rather than after full response processing.

Example: session created using `examplecookie`:

```
upstream backend {
    zone backend 1m;
    server backend1.example.com:8080;
    server backend2.example.com:8080;
```

```
    sticky learn
        create=$upstream_cookie_examplecookie
        lookup=$cookie_examplecookie
        zone=client_sessions:1m;
}
```

`learn` with `remote_action` (PRO 1.8.0+)

Use `remote_action` and `remote_result` to manage session IDs with an external session store. The shared memory zone acts as a cache; the external store is authoritative. `create` is not compatible with `remote_action`.

Sessions expire after `timeout` (default: 1 hour), regardless of `remote_action`.

By default, Angie refreshes session TTL on each use. Use `norefresh` to disable that.

`zone` is optional with `remote_action`. Without it, Angie always queries the external store.

Basic flow:

- Extract session ID from the first non-empty `lookup` variable. If none, fall back to standard load balancing.
- If `zone` is set and session exists, use it and stop.
- If no session or no zone, pick a server and make an HTTP subrequest to the `remote_action` endpoint with:
  - session ID (*$sticky_sessid*);
  - server ID from `sid=` or from *$sticky_sid*.

  Send these as HTTP headers (via *proxy_set_header*).
- The remote store responds:
  - 200/201/204 confirms the session; cache it if `zone` is set.
  - 409 signals conflict (if `zone` is set) — session linked to another server. Use `remote_result` to extract the corrected server ID.
  - Other status codes or missing server ID — fallback to original server.

`remote_result` uses `upstream_http_*` variables to read headers from the remote store's response.

Example: session ID comes from `$cookie_bar`, confirmed via `$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;
        }
    }
}
```

The following is a simplified configuration example. The remote store returns the session ID in the `X-Sid` header and so confirms or overrides Angie's choice:

```
http {

    proxy_cache_path c1 keys_zone=s1:1m;

    upstream tc_0 {
        server 10.0.0.1 sid=web-server-01;
        server 10.0.0.2 sid=web-server-02;

        sticky learn
            lookup=$arg_id
            remote_action=@create_session
            remote_result=$upstream_http_x_sid;
    }

    server {
        listen 127.0.0.1:8080;

        location / {
            proxy_pass http://tc_0/;
        }

        # Request to the remote session store
        location @create_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://session_backend;

            proxy_connect_timeout 1s;
            proxy_read_timeout    1s;

            proxy_cache        s1;
            proxy_cache_valid  200 1d;
            proxy_cache_key    "$scheme$proxy_host$request_uri$sticky_sessid";
        }
    }
}
```

Example response:

```
HTTP/1.1 200 OK
...
X-Sid: web-server-01
X-Session-Backend: backend-pool-1
```

Resulting Angie variables:

- `$upstream_http_x_sid` → `web-server-01`
- `$upstream_http_x_session_backend` → `backend-pool-1`

`remote_result` will use `web-server-01` to select the matching `sid`.

The `sticky` directive respects upstream server states:

- Servers marked `down` or failing are excluded.
- Servers over `max_conns` limit are skipped.
- `drain` servers (PRO) may still be selected for new sessions in `sticky` mode when identifiers match.
- Recovered servers are reused automatically.

You can further adjust behavior using *sticky_secret* and *sticky_strict*. If stickiness fails and `sticky_strict` is off, fallback balancing is used; if on, the request is rejected.

Each `zone` used in `sticky` must be exclusive to a single `upstream`. Zones cannot be shared across multiple `upstream` blocks.

sticky_secret

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 {
    zone backend 1m;
    server backend1.example.com:8080;
    server backend2.example.com:8080;

    sticky cookie cookie_name;
    sticky_secret my_secret.$remote_addr;
}
```

The salt is appended to the value being hashed; to verify the hashing mechanism independently:

```
$ echo -n "<VALUE><SALT>" | md5sum
```

sticky_strict

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 {
    zone backend 1m;
    server backend1.example.com weight=5;
    server 127.0.0.1:8080       max_fails=3 fail_timeout=30s;
    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. The distribution is *smooth*: a higher-weight server's requests are spread across the rotation rather than sent in a single burst.

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.

Note

By default, a server that fails an occasional attempt without reaching *max_fails* temporarily receives a smaller share of requests, regaining its full share over the requests that follow. This differs from *slow_start*, which ramps a server back up only after it has been marked unavailable and has recovered.

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.

Note

The zone's content is only preserved on reload when the configured `size` is unchanged. Any size change — increase or decrease — causes the zone to be re-created empty.

Note

Upstream metrics are collected only when this zone is configured. Without it, the group does not appear in */status/http/upstreams/<upstream>*, the *HTTP Upstreams Widget*, or *Prometheus* output, and no warning is logged. See *Example configuration*.

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 previously 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 is not found in the cache, and the request is passed to the upstream server.
- `BYPASS`: The cache is bypassed, and the request is passed directly to the upstream server.
- `EXPIRED`: The cached response is stale, and a new request is passed to the upstream server to update the content.
- `STALE`: The cached response is stale, but is still served to clients until the content is eventually updated from the upstream server.
- `UPDATING`: The cached response is stale, but is still served to clients while the currently ongoing update from the upstream server is in progress.
- `REVALIDATED`: The cached response is stale, but was successfully revalidated and does not need to be updated from the upstream server.
- `HIT`: The response was taken from the cache.

If the request bypassed the cache without accessing it, the variable is not set.

`$upstream_cache_key`

Added in version 1.11.0.

contains the cache key used for the request.

`$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`

keeps 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>`

keep 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_request_method`

Added in version 1.11.0.

request method used for the upstream request. It can differ from the client request method when caching converts `HEAD` to `GET` or when *proxy_ method* is set.

`$upstream_queue_time`

keeps time the request spent in the *queue* before the next server selection; the time is kept in seconds with millisecond resolution. Times of several 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 an upstream where sticky is not enabled.
`NEW`	Request does not contain sticky information.
`HIT`	Request with sticky information sent to the desired server.
`MISS`	Request with sticky information sent to a server selected by the load balancing algorithm.

Statuses from several connections are separated by commas and colons like addresses in the *$upstream_addr* variable.

`$upstream_trailer_<name>`

keeps fields from the end of the response obtained from the upstream server.

Upstream Probe

The module implements active health probes for *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)

Syntax	`upstream_probe` *name* [`uri=`*address*] [`port=`*number*] [`interval=`*time*] [`method=`*method*] [`test=`*condition*] [`essential` [`persistent`]] [`fails=`*number*] [`passes=`*number*] [`max_body=`*size*] [`mode=always` | `idle` | `onfail`];
Default	—
Context	location

Defines an active health probe for servers within the *upstream* groups that are specified in *proxy_pass*, *uwsgi_pass*, and similar directives in the same `location` context as the `upstream_probe` directive. Angie regularly performs requests according to the specified parameters to each server in the upstream group.

A server passes the probe if the request to it succeeds, considering all parameter settings of the `upstream_probe` directive and all parameters that control how upstreams are used by the `location` context where it is defined. This includes the *proxy_next_upstream* and *uwsgi_next_upstream* directives, etc., as well as *proxy_set_header* and so on.

To use 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 appended to the argument of *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 request. By default — `GET`.
`test`	The condition to be checked during the request; defined as a string with variables. If variable substitution yields `""` or `"0"`, the probe fails.
`essential`	If set, the initial state of the server is subject to verification and client requests are not forwarded to it until the probe is passed.
`persistent`	Setting this parameter requires enabling `essential` first; `persistent` servers that were working prior to a *configuration reload* start receiving requests without being required to pass this probe first.
`fails`	Number of consecutive failed requests that renders the server unhealthy. By default — 1.
`passes`	Number of consecutive successful requests that renders the server healthy. By default — 1.
`max_body`	Maximum amount of memory for the response body. By default — `256k`.
`mode`	Probe mode, depending on the servers' health: • `always` — servers are probed regardless of their state; • `idle` — probes affect unhealthy servers and servers where `interval` has elapsed since the last client request. • `onfail` — only unhealthy servers 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 server won't receive client requests until it passes *all* `essential` probes configured for it (skipping `persistent` ones if the configuration was reloaded and the server was deemed healthy prior to that). If there are no such probes, the server is considered healthy.
- The server is considered unhealthy and won't receive client requests if *any* of the probes configured for it hits its `fails` threshold or the server itself reaches the *max_fails* threshold.
- For an unhealthy server to be considered healthy again, *all* probes configured for it must reach their respective `passes` thresholds; after that, the *max_fails* threshold is considered.

Built-in Variables

The `http_upstream_probe` module supports the following built-in variables:

`$upstream_probe` (PRO)

Name of the currently active *upstream_probe*.

`$upstream_probe_body` (PRO)

Server 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.

Note

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 on` | `off`;
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. Sending to the client is performed in parallel: filled buffers are passed for sending, but their total size is limited by *uwsgi_busy_buffers_size*. If a buffer is not completely filled, it is not passed for sending unless it contains the last part of the response. Therefore, buffered reading is not suitable when you need immediate delivery of every few bytes. 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 immediately as it is received. Angie works in a "read — send" loop and does not wait for the buffer to fill completely: for example, 10 bytes read from a 4K buffer are sent right away. At the same time, if the entire response fits into the buffer, Angie can read it in full. The maximum size of the data that Angie can receive from the server at a time is set by the *uwsgi_buffer_size* directive. With `off`, *uwsgi_limit_rate* does not work.

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.

Warning

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.

Warning

Cache metadata is stored in shared memory. Manually deleting cache files does not reset the counters and may lead to unpredictable behavior. To completely reset the cache, stop the server, delete the cache directory, and start again.

Note

Third-party cache purge modules (e.g., Cache Purge) only delete files but do not reset the *uwsgi_cache_min_uses* counter. The directive is intended to protect the cache from pollution by infrequent requests, and resetting the counter on purge could negatively impact performance.

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 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 *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. Cache metadata is stored in shared memory.

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 threshold value for cache size
`min_free`	minimum threshold value for free space on the filesystem with cache
`manager_files`	maximum number of cache items deleted in one iteration Default: `100`
`manager_threshol`	limits the time of one iteration Default: `200` milliseconds
`manager_sleep`	time for which a pause is maintained between iterations Default: `50` milliseconds

One minute after Angie starts, a special **cache loader** process is activated, which loads information about previously cached data stored on the filesystem into the cache zone. Loading also occurs in iterations.

`loader_files`	maximum number of cache items to load in one iteration Default: `100`
`loader_threshold`	limits the time of one iteration Default: `200` milliseconds
`loader_sleep`	time for which a pause is maintained between iterations 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. 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 method has lower priority than setting directive parameters.

To minimize the number of requests 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, it can be specified to cache any responses using the `any` parameter:

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

Note

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

- The `X-Accel-Expires` header field sets caching time of a response in seconds. The value *0* 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, caching parameters are determined by the `Expires` or `Cache-Control` header fields.
- A response with the `Set-Cookie` header field will not be cached.
- A response with the `Vary` header field with the special value "*" will not be cached. A response with the `Vary` header field with another value will be cached taking into account the corresponding request header fields.

Processing of one or more of these 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

Configures termination of all connections to the proxied server if it has been removed from the group or marked as permanently unavailable as a result of a *reresolve* process or *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 on` | `off`;
Default	`uwsgi_force_ranges off;`
Context	http, server, location

Enables byte-range support for both cached and uncached responses from a uwsgi server regardless of the presence 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 `Date`, `Server`, `X-Pad`, 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, conversely, 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; variables can be used.

`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` *number*;
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` *number*;
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* group:

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

Note

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

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

`error`	are always considered unsuccessful attempts, even if they are not specified in the directive
`timeout`	
`invalid_header`	
`http_500`	are considered unsuccessful attempts only if they are specified in the directive
`http_503`	
`http_429`	
`http_403`	are never considered unsuccessful attempts
`http_404`	

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;
#    ...
}
```

In the standard `uwsgi_params` file, `REQUEST_METHOD` is set to `$upstream_request_method`.

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. 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 specified after the word `unix` and enclosed in colons:

```
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*.

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*.

Note

If `uwsgi_pass` is specified in a `location` with a trailing slash in the prefix (for example, `location /name/`), and the *auto_redirect* directive is set to `default`, requests without a trailing slash will be redirected (`/name -> /name/`).

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

Enables or disables passing of header fields from the original request to the uwsgi server. See also the *uwsgi_pass_request_body* directive.

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 request body is fully *read* from the client before sending the request to the 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 has already started sending the request body.

When HTTP/1.1 chunked transfer encoding is used to send the original request body, then 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.

`off`	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.

The value `store:scheme:id` can be specified instead of the file, which is used to load a secret key with a specified id and OpenSSL provider registered URI scheme, such as pkcs11.

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.

Warning

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 a secured uwsgi server.

Note

The directive is supported when using OpenSSL 1.0.2 or higher. To configure TLS 1.3 ciphers in OpenSSL, use the `ciphersuites` command.

Multiple *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.

Warning

Note that reconfiguring 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 name 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 name from the *uwsgi_pass* directive 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

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 secured 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 uwsgi HTTPS 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 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 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 build parameter `--http-uwsgi-temp-path`)
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 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 repositories, the module is included in the build.

Configuration Example

```
server {
    listen 443 ssl;

    http2 on;

    ssl_certificate server.crt;
    ssl_certificate_key server.key;
}
```

Note

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 the RFC 9113, Appendix A blacklist and be supported by clients.

Directives

http2

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 head-of-line 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

Provides HTTP/3 protocol support for client connections, as well as for connections with proxied servers configured using the following *Proxy* module 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 repositories, 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';
        }
    }
}
```

Note

Note that accepting HTTP/3 connections over TLS requires the TLSv1.3 protocol support, which is available since OpenSSL version 1.1.1.

0-RTT support requires OpenSSL 3.5.1 or higher. Alternatively, BoringSSL, LibreSSL, or QuicTLS can be used to build and run this module.

Before version 1.29.1, 0-RTT support could not be enabled with OpenSSL regardless of the *ssl_early_data* directive value.

For HTTP/3 requests, if the `Host` header is not passed, the `$http_host` variable is initialized from the `:authority` pseudo-header.

Also, the `reuseport` option can only be specified in one of the `listen ... quic` directives on a server. All other `listen ... quic` directives must be specified without it.

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.

Warning

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 connection IDs that 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.

Note

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.

Note

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`.

> **Note**
>
> 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;
```

Using `aio on` requires building with the `--with-file-aio` configuration parameter. Using `aio threads` requires building with the `--with-threads` 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

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

Added in version 1.10.0.

Changed in version 1.10.1.

Syntax	`client { ... }`
Default	—
Context	http

Creates a special `client` context for processing internal HTTP requests that Angie performs on its own without external client involvement.

The `client` context isolates service traffic from various Angie modules from user traffic, allowing additional control over it. Within this context, only named locations (with the `@` prefix) can be defined; they are not accessible for external HTTP requests and can only be called programmatically through internal server mechanisms.

The `client` context is used for:

- sending requests to the certificate authority in the *ACME* module via the predefined `location @acme`, which can be additionally configured using directives from the *Proxy* module;
- requests to the Docker API in the *Docker* module via the predefined `location @docker_events` and `@docker_containers`, which can be additionally configured using directives from the *Proxy* module;
- health probes of proxied servers via *upstream_probe (PRO)*;

- *sticky learn* mode with `remote_action` in the stream *Upstream* module.

Support for multiple `client` blocks allows grouping common settings for multiple `location` blocks within each block, which helps avoid configuration duplication.

Directives specified in each `client` block are inherited only by `location` blocks explicitly declared within it. In particular, this is why they do not affect the configuration of other modules that implicitly use the `client` block for outgoing requests (for example, *ACME* or *Docker*).

Example of using multiple `client` blocks with settings inheritance:

```
client {

    proxy_set_header Host docker.example.com;
    proxy_set_header Authorization "Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==";

    location @docker_events {

    }

    location @docker_containers {

    }
}

client {

    proxy_method GET;
    proxy_set_header Host backend.example.com;
    proxy_set_header X-Real-IP $remote_addr;

    location @health_check {

        proxy_pass http://upstream-server/health;
    }
}
```

Note

The same directives are allowed here as in regular `location` blocks, but only content handlers (such as js_content or *autoindex*) and variable handlers (such as *map*), as well as directives that generate requests themselves, like `upstream_probe`, actually work.

Directives that operate at other *request processing stages* (such as *limit_req*, *auth_request*, *try_files*, image filters, XSLT, etc.) do not work here.

client_body_buffer_size

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

Sets the buffer size for reading the client request body. If the request body is larger than the buffer, the whole body or only its part is written to a *temporary file*. By default, the buffer size is equal to two memory pages. On x86, other 32-bit platforms, and x86-64, this is 8K. On other 64-bit platforms, it is usually 16K.

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 to save the entire client request body to 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 *Perl* module.

`on`	temporary files are not removed after request processing
`clean`	allows 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 to save the entire client request body in a single buffer. The directive is recommended when using the *$request_body* variable to reduce 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 build option `--http-client-body-temp-path`)
Context	http, server, location

Defines a directory for storing temporary files with 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. See the *Virtual server selection* section for details.

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)	on 32-bit platforms
`512` (bytes)	on 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 is recommended 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 path are allowed and not checked. This is the default behavior.
`on`	If any component of the path is a symbolic link, access to the file is denied.
`if_not_owner`	Access to the file is denied if any component of the path is a symbolic link, and the link and the object it points to have different owners.
`from=`*part*	When checking symbolic links (parameters `on` and `if_not_owner`), all path components are usually checked. It is possible to skip checking symbolic links in the initial part of the path by additionally specifying the `from=part` parameter. In this case, symbolic links are checked only starting from the path component that follows the specified initial part. If the value is not an initial part of the checked path, the path is checked entirely, as if this parameter were not specified at all. If the value completely matches the file name, symbolic links are not checked. Variables can be used in the parameter value.

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

The `on` and `if_not_owner` parameters add processing overhead.

On systems that do not support opening directories for search only, using these parameters requires worker processes to have read permissions for all directories being checked.

Note

The *AutoIndex*, *Random Index*, and *DAV* modules currently ignore this directive.

early_hints

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

Defines conditions under which the "103 Early Hints" response will be passed to a client. The response can be returned by proxied and gRPC backends. If at least one value of the string parameters is not empty and is not equal to `0` then the response will be passed:

```
map $http_sec_fetch_mode $early_hints {
    navigate $http2$http3;
}

server {
    ...
    location / {
        early_hints $early_hints;
        proxy_pass http://example.com;
    }
}
```

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. The *uri* value can use 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 syntax like `=response`, 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 pass the code it returns:

```
error_page 404 = /404.php;
```

If there is no need to change the URI and method during internal redirect, it is possible to pass error processing into a named `location`:

```
location / {
  error_page 404 = @fallback;
}

location @fallback {
  proxy_pass http://backend;
}
```

Note

If an error occurs during the processing of uri, the response with the 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 response 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 Angie ignores header fields with invalid names. Valid names are composed of English letters, digits, hyphens, and possibly underscores (as controlled by the *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 *SSI* module, by the *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;
}
```

Because the 404 error is returned in the context of a `location` with the `internal` directive, external requests can be redirected to a different location. This allows using the same prefix for both external and internal requests, but with different processing, for example:

```
location /path {

    internal;
    error_page 404 =@external;

    proxy_pass https://internal;
}

location @external {

    proxy_pass https://external;
}
```

Here, an external request `GET /path` will be proxied to `https://external/path`, while the same internal request will be proxied to `https://internal/path`.

Note

To prevent looping that can occur with incorrect configurations, the number of internal redirects is limited to ten. When this limit is reached, the 500 (Internal Server Error) 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.

Periodic closing of connections is necessary to free per-connection memory allocations. Therefore, using too high maximum number of requests could result in excessive memory usage and is 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 second, *optional*, parameter sets a value in the `Keep-Alive: timeout=time` header field in the response. The 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* parameter 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 makes the `HEAD` method also allowed. Access to other methods can be limited using the *Access* and *Auth Basic* module directives:

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

Note

The restriction in this example applies 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`	Angie will *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`	Angie will always wait for and process additional client data.
`off`	Angie will not wait for more data and will close the connection immediately. This behavior breaks the protocol and should not be used under normal circumstances.

To control closing of HTTP/2 connections, the directive must be specified at 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, this 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. The "wait-read-ignore" cycle is repeated, but no longer than specified by the *lingering_time* directive.

During graceful shutdown, client keepalive connections are closed only when they have been idle for at least the time specified in `lingering_timeout`.

Note

In nginx, the analogous directive is called keepalive_min_timeout.

listen

Changed in version 1.10.0.

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 the listening 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 only *port*, can be specified. When some parts are omitted, the following rules apply:

- 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 `server` block for that port becomes the default server 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`	indicates that all connections accepted on this listening socket 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`	indicates that all connections accepted on this listening socket should use the PROXY protocol.

The `listen` directive can also specify several additional parameters specific to socket-related system calls. These parameters can be specified in any `listen` directive, but only once for a given listening socket:

`setfib=`*number*	sets the 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. **Warning** Do not enable "TCP Fast Open" 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+. 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`	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`	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+. **Warning** Inappropriate use of the `reuseport` parameter may have security implications.
`multipath`	enables accepting connections via Multipath TCP (MPTCP), supported in the Linux kernel since version 5.6. This parameter is **incompatible** with `quic`.
`so_keepalive=on` | `off` | [`keepidle`]:[`keepint`	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,

NetBSD, Dragonfly, FreeBSD, and macOS), 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 (prefix locations). Among them, the location with the longest matching prefix is selected and remembered.

Note

For case-insensitive operating systems such as macOS, prefix string matching is case insensitive. However, matching is limited to single-byte locales.

Then regular expressions are checked in the order of their appearance in the configuration file. The search stops after the first match, and the corresponding configuration is used. If no match with a regular expression is found, then the configuration of the prefix location remembered earlier is used.

With some exceptions mentioned below, `location` blocks can be nested.

Regular expressions can create capture groups that can later be used with other directives.

If the longest matching prefix location has the `^~` modifier, then regular expressions are not checked.

Also, using the = modifier, it is possible to define an exact match of URI and location. If an exact match is found, the search terminates. For example, if a `/` request happens frequently, defining `location =/` will speed up the processing of these requests, as the search terminates after the first comparison. Such a location cannot contain nested locations, as it defines an exact match.

Example:

```
location =/ {
    #configuration A
}

location / {
    #configuration B
}

location /documents/ {
    #configuration C
}

location ^~/images/ {
    #configuration D
}

location ~*\.(gif|jpg|jpeg)$ {
    #configuration E
}
```

- A `/` request will match configuration A,
- an `/index.html` request will match configuration B,
- a `/documents/document.html` request will match configuration C,
- an `/images/1.gif` request will match configuration D,
- and a `/documents/1.jpg` request will match 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 a URI that has no trailing slash but otherwise matches the prefix exactly, a permanent redirect with code 301 is returned, pointing to the requested URI with a 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 are only intended 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 {
   #...
}
```

Warning

A combined `location` can't have a space between the matching expression modifier and the expression itself. Proper form: `location ~*/match(ing|es|er)$ ....`

Note

Currently, a combined `location` cannot **immediately** contain `proxy_pass` 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, 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_events

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

Enables the use of kernel events to validate *open_file_cache* elements. This directive works with the *kqueue* method only. Note that only NetBSD 2.0+ and FreeBSD 6.0+ support events for arbitrary file system types; other operating systems support events only for essential file systems such as UFS or FFS.

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 the specified number of bytes 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 files.

On Linux, the `posix_fadvise(0, 0, 0, POSIX_FADV_SEQUENTIAL)` system call is used, and so the size parameter is ignored.

On FreeBSD, the `fcntl(O_READAHEAD,` size `)` system call, supported since FreeBSD 9.0-CURRENT, is used.

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 for it with a timeout value of 0. When the socket is closed, TCP RST is sent to the client, and all memory associated with this socket is released. This helps avoid keeping an already closed socket in the FIN_WAIT1 state with filled buffers for a long time.

Note

keep-alive connections are closed normally when they time out.

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.

Note

Prefer a local trusted resolver such as `127.0.0.53` (systemd-resolved) over a public one (e.g. `8.8.8.8`). Public resolvers expose DNS queries to third parties and increase susceptibility to cache-poisoning attacks.

Note

The directive value is inherited by nested blocks and can be overridden in them if necessary. Within a single block, the directive may only be specified once. If it is repeated, the last definition takes effect.

By default, Angie caches answers using the TTL value of a response. If the `resolver` directive is not specified and no dynamic DNS queries are performed (for example, when using fixed names in *Proxy* without variables), specifying a resolver is not required: names will be resolved at startup using the system resolver. The optional `valid` parameter allows overriding this:

`valid`	*optional* parameter allows overriding the response cache validity period

```
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 in the specified zone, exposing them in */status/resolvers/<zone>*, the *DNS Resolvers Tab*, and *Prometheus* output. Without it, these metrics are not collected and no warning is logged

Tip

To prevent DNS spoofing, it is recommended to use DNS servers in a properly secured trusted local network.

Tip

When running in Docker, use the corresponding 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;
```

error_log_user_tag

Syntax	`error_log_user_tag` *value*;
Default	—
Context	http, server, location, limit_except

Adds a request-specific tag to error log records. The *value* is a *complex value* and can use variables. The directive can be specified multiple times to add multiple tags. Tags can be matched with `filter=tag:` in *error_log*.

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 the *Access*, *Auth Basic*, or *Auth Request* modules allow access.

```
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 the `NOTE_LOWAT` flag of the *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's parameter is set to *$hostname*, the machine name is used.

An empty server name can also be specified:

```
server {
    server_name www.example.com "";
}
```

When searching for a virtual server by name, if the name matches more than one of the specified variants (for example, both a wildcard name and regular expression match), the first matching variant will be chosen, in the following order of priority:

- exact name;
- longest wildcard name starting with an asterisk, e.g. `*.example.com`;
- longest wildcard name ending with an asterisk, e.g. `mail.*`;
- first matching regular expression (in order of appearance in the configuration file), including an empty name.

Warning

To use `server_name` with TLS, TLS connection termination is required. This directive matches against the `Host` in the 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 set by the *server_name* directive is used
`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 the 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.

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*[:*number*];
Default	—
Context	server, location, if in location

Allocates a shared memory zone for collecting */status/http/location_zones/<zone>* and */status/http/server_zones/<zone>* metrics.

Several `server` contexts can share the same zone for data collection; the special value `off` disables data collection in nested `location` blocks.

The syntax with a single *zone* value combines all metrics for the current context into one shared memory zone:

```
server {

    listen 80;
    server_name *.example.com;

    status_zone single;
    # ...
}
```

The alternative syntax allows setting the following 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.
number (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.

Note

These metrics are collected only when `status_zone` is set. Without it, the server or location does

not appear in */status/http/server_zones/<zone>*, */status/http/location_zones/<zone>*, the *HTTP Zones Widget*, or *Prometheus* output, and no warning is logged. See *Example configuration*.

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 location's 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 be a URI for an internal redirect, a reference to a named `location` (e.g., `@drupal`), or a response code in the form `=code` (e.g., `=404`):

```
location / {
  try_files $uri $uri/index.html $uri.html =404;
}
```

It should be noted that excessive use of the `try_files` directive increases the number of system calls, which can negatively impact performance.

Thus, `try_files` should not be used to replicate behavior that is effectively the default behavior, for example:

```
location /bad_pattern {

      # try_files $uri $uri/ =404;  # not recommended!
}
```

Also, `try_files` should not be used solely for redirecting when a file is absent. The reason is that the `try_files` directive has two peculiarities:

- First, it checks the existence of each file, which increases system load.
- Second, any file opening errors (e.g., `too many open files`, permission errors) are also treated as file absence and trigger a fallback to the backup handler, which can mask 5xx errors with successful responses and lead to incorrect caching.

Thus, in practice, the following problematic construction can be encountered:

```
location / {
   try_files $uri $uri/ @drupal;  # not recommended!
}
```

The problem here is that the only purpose is redirection. Using `try_files` leads to the disadvantages listed above, but provides no benefits, since checking for file existence is not needed. The correct solution is to use the *error_page* directive, which does not have these disadvantages:

```
error_page 404 = @drupal;
log_not_found off;
```

In contrast, in the following example:

```
location ~ \.php$ {
  try_files $uri @drupal;

  fastcgi_pass ...;

  fastcgi_param SCRIPT_FILENAME /path/to$fastcgi_script_name;
```

```
#  ...
}
```

The `try_files` directive checks for the existence of the PHP file before passing the request to the FastCGI server configured in the same block; here the use of `try_files` is justified.

Example of use when proxying to Mongrel:

```
location / {
  try_files /system/maintenance.html
            $uri $uri/index.html $uri.html
            @mongrel;
}

location @mongrel {
  proxy_pass http://mongrel;
}
```

Example of use with Drupal/FastCGI:

```
location / {
  error_page 404 = @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
}
```

Example of use with Wordpress and Joomla:

```
location / {
  error_page 404 = @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 complete mapping table is distributed with Angie and is located in the `conf/mime.types` file.

To make a particular `location` return the "application/octet-stream" MIME type for all responses, 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 discussed *separately*.

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 discussed *separately*.

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 are subject to the *ignore_invalid_headers* directive.

If the directive is specified at 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 discussed *separately*.

variables_hash_max_size

Changed in version 1.11.0.

Syntax	`variables_hash_max_size` *size*;
Default	`variables_hash_max_size 2048;`
Context	http

Sets the maximum size of the variables hash table. The details of setting up hash tables are discussed *separately*.

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 *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>`

Changed in version 1.11.0: In HTTP/3 requests, `$http_host` is initialized from the `:authority` pseudo-header if the `Host` header was not passed by the client.

arbitrary request header field; the last part of the variable name corresponds to 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

`$is_request_port`

`:` if the *$request_port* value is non-empty, 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

`$nginx_version`

nginx version

`$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 name 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, enable *client_body_in_file_only*. When passing the name of a temporary file in a proxied request or in a request to a FastCGI/uwsgi/SCGI server, the passing of the request body itself should be disabled with 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_port`

in this order of precedence: port number from the authority component of the request URI, or port number from the "Host" request header field

`$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), never modified during request processing; see *$uri* for the current (potentially rewritten) URI

`$scheme`

request scheme, "http" or "https"

`$sent_body`

Added in version 1.11.0.

response body of a subrequest or external request when it is stored in memory; otherwise an empty string

`$sent_http_<name>`

arbitrary response header field; the last part of the variable name corresponds to 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 last part of the variable name corresponds to the field name converted to lower case with dashes replaced by underscores

`$server_addr`

address of the server which accepted a request

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_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 request, *normalized*

The value of `$uri` may change during request processing, e.g. when rewriting with *rewrite*, 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	stream, server

Allows access for the specified network or address. If the special value `unix:` is specified, allows access for all UNIX domain sockets.

deny

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

Denies access for the specified network or address. If the special value `unix:` is specified, denies access for all UNIX domain sockets.

ACME

Added in version 1.10.0.

Allows automatic certificate acquisition using the ACME protocol for servers defined in the `stream` context.

When building from source the module is not built by default; it must be enabled with the build parameter `--with-stream_acme_module` (also requires `--with-http_acme_module`). In packages and images from our repositories the module is included in the build.

Note

For correct operation, the `stream` block must be located after the `http` block. This is because the stream module uses client definitions created during HTTP configuration parsing.

Configuration Example

For configuration examples and setup instructions, see the *ACME in the Stream Module* section.

Directives

acme

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

For all domains specified in *server_ name* directives in all *server* blocks that reference an *ACME client* from the HTTP module with the given *name*, a single certificate will be obtained; if the `server_name` configuration changes, the certificate will be updated to account for the changes.

On each Angie startup, new certificates are requested for all domains that lack a valid certificate. Possible reasons include certificate expiration, missing files or inability to read them, and changes in certificate settings.

Note

Currently, domains specified via regular expressions are not supported and will be skipped.

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

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

```
server {

    listen 12345 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;
}
```

Embedded 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*.

Note

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

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.
`volatile`	indicates that the variable is not cacheable.

Example:

```
geo $country {
    default         ZZ;
    include         conf/geo.conf;
    delete          127.0.0.0/16;

    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 should be enabled with the `--with-stream_geoip_module` build option.

Note

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".
`$geoip_dma_code`	DMA region code in the US (also known as "metro code"), according to the geotargeting in Google AdWords API.
`$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".

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.

Warning

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 the time interval 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 buffer 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, a buffer size of `64K` bytes and compression level `1` are used. The data is compressed in atomic blocks, and at any time the log file can be decompressed or read by the `"zcat"` utility.

Example:

```
access_log /path/to/log.gz basic gzip flush=5m;
```

Note

For gzip compression support, Angie must be built with the zlib library.

Variables can be used in the file path, 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;
- buffering does not work;
- the file is opened for each log write and closed immediately after writing. However, since descriptors of frequently used files can be stored in a cache, writing may continue to the old file during log rotation for the time specified by the *valid* parameter of the *open_log_file_cache* directive.

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

Specifies the log format, for example:

```
log_format proxy '$remote_addr [$time_local] '
                 '$protocol $status $bytes_sent $bytes_received '
                 '$session_time "$upstream_addr" '
                 '"$upstream_bytes_sent" "$upstream_bytes_received" "$upstream_
↪connect_time"';
```

The `escape` parameter allows setting `json` or `default` character escaping in variables; by default, `default` is used. The `none` value disables character escaping.

When using `default`, characters """, "\", and characters with values less than 32 or greater than 126 are escaped as "\xXX". If the variable value is not found, a hyphen "-" will be logged.

When using `json`, all characters not allowed in JSON strings are 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*]; `open_log_file_cache off;`
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 the cache; when the cache overflows, the least recently used (LRU) descriptors are closed.
`inactive`	Sets the time after which a cached descriptor is closed if there were no accesses during this time; by default, 10 seconds.
`min_uses`	Sets the minimum number of file uses during the time defined by the `inactive` parameter for the descriptor to remain open in the 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

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, the module must be enabled with the build parameter `--with-stream_mqtt_preread_module`. In packages and images from our repositories, the module is included in the build.

Configuration Example

Choosing a server in a group by 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 parameter is enabled (`on`), the variables listed below are populated in the context where it is specified.

Built-in Variables

For detailed description of value semantics, see the MQTT protocol specification versions 3.1.1 and 5.0.

`$mqtt_preread_clientid`

Unique client identifier.

`$mqtt_preread_username`

Optional username.

Pass

Allows passing the accepted connection directly to any configured listening socket in *HTTP*, *Stream*, or *Mail* modules.

The module enables selective SSL termination based on SNI.

Configuration Example

After the `stream` module handles the SSL/TLS termination, it forwards the connection to the `http` module:

```
stream {

    server {

        listen 8000 default_server;
        ssl_preread on;
        # ...
    }

    server {

        listen 8000;
        server_name foo.example.com;
        pass 127.0.0.1:8001; # to HTTP
    }

    server {

        listen 8000;
        server_name bar.example.com;
        # ...
    }
}

http {

    server {

        listen 8001 ssl;
        # ...

        location / {

            root html;
        }
    }
}
```

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.

Note

The kernel routing table should also be configured to intercept network traffic from the proxied 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_protocol_version

Syntax	`proxy_protocol_version 1 | 2;`
Default	`proxy_protocol_version 1;`
Context	stream, server

Sets the PROXY protocol version used for connections to a proxied server. The setting is effective when *proxy_protocol* is enabled. Version 2 allows sending TLVs configured by the *proxy_protocol_tlv* directive.

proxy_protocol_tlv

Added in version 1.11.0.

Syntax	`proxy_protocol_tlv` *name value*;
Default	—
Context	stream, server

Adds a TLV to the PROXY protocol v2 header sent to a proxied server. The *value* can contain variables. The *name* can be a TLV type name or its numeric value; in the latter case, the value is specified in hexadecimal and must start with *0x*. For SSL TLVs, use the `ssl_` prefix; the special `ssl_verify` name sets the verify field of the SSL TLV. The directive is used only with *proxy_protocol_version* set to `2`.

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.

`off`	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.

When *proxy_ssl_ntls* is enabled, the directive takes 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

The value `store:scheme:id` can be specified instead of the `file`, which is used to load a secret key with a specified *id* and OpenSSL provider registered URI *scheme*, such as pkcs11.

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.

When *proxy_ssl_ntls* is enabled, the 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.

Warning

The `proxy_ssl_ciphers` directive does *not* configure ciphers for TLS 1.3 when using OpenSSL. To configure TLS 1.3 ciphers with OpenSSL, use the *proxy_ssl_conf_command* directive, which was added for 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.

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.

Note

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.

Warning

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 name from the address specified by the *proxy_pass* directive is used.

proxy_ssl_ntls

Syntax	`proxy_ssl_ntls on | off`;
Default	`proxy_ssl_ntls off`;
Context	stream, server

Enables client-side support for NTLS when 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;
}
```

Note

Angie must be built using the `--with-ntls` configuration parameter, with the corresponding SSL library with NTLS support

```
./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

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 specified by the *proxy_ssl_name* directive through the Server Name Indication TLS extension (SNI, RFC 6066) when 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 a 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.

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 connection, so if the client simultaneously opens two connections, the overall rate will be twice as much as the specified limit.

The parameter value can contain variables. This may be useful in cases where the 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, the module must 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 configuration 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`, the variables listed below will be populated in the context where it is specified.

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

Allows changing the client address and port to those passed 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

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 generates variables for A/B testing, canary releases, and other scenarios that route a specific 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 *$variable* by hashing the *string*; variables in the *string* are substituted, the result is hashed, and the hash value is used to select the string value of the *$variable*.

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 appear at the end; hashes that don't fall into other buckets are mapped to its assigned value.

Example:

```
split_clients "${remote_addr}AAA" $variant {
              0.5%                .one;
              2.0%                .two;
              *                   "";
}
```

Here, after substitution in the `$remote_addrAAA` string, the hash values are distributed as follows:

- values from 0 to 21474835 (0.5%) yield `.one`

- values from 21474836 to 107374180 (2%) yield `.two`
- values from 107374181 to 4294967295 (all others) 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.

Note

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.

Note

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_compression

Syntax	`ssl_certificate_compression on | off;`
Default	`ssl_certificate_compression off;`
Context	stream, server

Enables TLS 1.3 compression of server certificates.

Note

The directive is supported when using OpenSSL 3.2 or higher; the list of supported compression algorithms is provided by the library.

Note

The directive is supported when using BoringSSL; the list of supported compression algorithms includes `zlib`.

If *ssl_stapling* is enabled, certificate compression is disabled.

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.

Note

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 `store:scheme:id` can be specified instead of the `file`, which is used to load a secret key with a specified *id* and OpenSSL provider registered URI *scheme*, such as pkcs11.

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.

Warning

The `ssl_ciphers` directive does *not* configure ciphers for TLS 1.3 when using OpenSSL. To configure 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.

Note

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.

Warning

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.

Warning

By default no parameters are set, and therefore DHE ciphers will not be used.

ssl_early_data

Added in version 1.9.0.

Syntax	`ssl_early_data on` | `off`;
Default	`ssl_early_data off;`
Context	stream, server

Enables or disables TLS 1.3 early data.

Note

The directive is supported when using OpenSSL 1.1.1 or higher or BoringSSL.

ssl_encrypted_hello_key

Added in version 1.11.0.

Syntax	`ssl_encrypted_hello_key` *file*;
Default	—
Context	stream, server

Specifies a file with an ECH private key and ECHConfigList in PEM format. The directive can be specified multiple times. Requires an OpenSSL or BoringSSL build with Encrypted Client Hello (ECH) support; otherwise it is not supported.

ssl_ecdh_curve

Syntax	`ssl_ecdh_curve` *curve*;
Default	`ssl_ecdh_curve auto;`
Context	stream, server

Specifies a curve for ECDHE ciphers.

Note

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.

Note

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	stream, 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	stream, 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	stream, 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

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;
```

Note

Angie must be built using the `--with-ntls` build option and linked 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

Enables the specified protocols.

Note

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) will be 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.

Warning

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 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

Note

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_sigalg`

returns the signature algorithm for the client certificate for an established SSL connection.

Note

The variable is supported only when using OpenSSL version 3.5 or higher. With older versions, the variable value will be an empty string.

Note

The variable is available only for new sessions.

`$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
```

Note

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
```

Note

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_encrypted_hello`

Added in version 1.11.0.

returns "1" if Encrypted Client Hello (ECH) is used, 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_sigalg`

returns the signature algorithm for the server certificate for an established SSL connection.

Note

The variable is supported only when using OpenSSL version 3.5 or higher. With older versions, the variable value will be an empty string.

Note

The variable is available only for new sessions.

SSL Preread

Enables extracting information from the ClientHello message without terminating 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 protocol 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 protocol 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

Provides context for describing groups of servers that can be used in 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

Describes 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 {
    zone backend 1m;
    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, connections are distributed between the servers using a weighted round-robin balancing method. In the above example, each 7 connections will be distributed as follows: 5 connections go to backend1.example.com:1935 and one connection to each of the second and third servers. The distribution is *smooth*: a higher-weight server's connections are spread across the rotation rather than sent in a single burst.

If an error occurs during communication with a server, the connection will be passed to the next server, and so on until all of the functioning servers will be tried. If communication with all servers fails, the connection will be closed.

Note

By default, a server that fails an occasional attempt without reaching *max_fails* temporarily receives a smaller share of connections, regaining its full share over the connections that follow. This differs from *slow_start*, which ramps a server back up only after it has been marked unavailable and has recovered.

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` directive in a group resolves to multiple servers, its `max_fails` setting applies to each server individually.

If an upstream contains only one server 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 attempts.
`max_fails=0`	Disables the accounting of attempts.

`fail_timeout=`*time* — sets the period of time during which a specified number of unsuccessful attempts to communicate with the server (*max_fails*) should happen to consider the server unavailable. The server then remains unavailable for the same amount of time before it is retried.

By default, this is set to 10 seconds.

Note

If a `server` directive in a group resolves to multiple servers, its `fail_timeout` setting applies to each server individually.

If an upstream contains only one server 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)	Marks the server as draining; this means it receives only requests from the sessions that were bound earlier with *sticky*. Otherwise it behaves similarly to `down`.

Warning

The `backup` parameter cannot be used along with the *hash* and *random* load balancing methods.

The `down` and `drain` parameters are mutually exclusive.

`resolve`	Enables monitoring changes to the list of IP addresses that corresponds to a domain name, updating it without a configuration reload. The group must reside in a *shared memory zone*; also, a *resolver* must be defined.
`service=`*name*	Enables resolving DNS SRV records and sets the service name. For this parameter to work, the *resolve* parameter must also be specified, without specifying the server port in the hostname. 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 to primary servers, and other records become backup servers. If `backup` is set with `server`, top-priority SRV records resolve to backup servers, and other records are ignored. • Weight is similar to the `weight` parameter of the `server` directive. If weight is set by both the directive and the SRV record, the weight set by the directive is used.

This example will look up the `_http._tcp.backend.example.com` record:

```
server backend.example.com service=http resolve;
```

`sid=`*id*	Sets the server ID in the group. If the parameter is not specified, the ID is set as a hexadecimal MD5 hash of the IP address and port or UNIX domain socket path.

`slow_start=`*time*	Sets the *time* for a server to recover its weight when returning to service with *round-robin* or *least_conn* load balancing methods. If the parameter is set and a server is again considered healthy after a failure according to *max_fails* and *upstream_probe (PRO)*, the server gradually recovers its designated weight over the specified time period. If the parameter is not set, in a similar situation the server immediately starts working with its designated weight.

Note

If only one `server` is specified in the upstream, `slow_start` has no effect and will be ignored.

state (PRO)

Syntax	`state` *file*;
Default	—
Context	upstream

Specifies the *file* where the upstream server list is persistently stored. When installing from our packages, a dedicated directory `/var/lib/angie/state/` (`/var/db/angie/state/` on FreeBSD) is created with appropriate permissions for storing such files, so you only need to add the filename in the configuration:

```
upstream backend {

    zone backend 1m;
    state /var/lib/angie/state/<FILE NAME>;
}
```

The server list format here is similar to `s_server`. The file contents change whenever servers are modified in the */config/stream/upstreams/* section via the configuration API. The file is read at Angie startup or configuration reload.

Warning

To use the `state` directive in an `upstream` block, there should be no `server` directives in it, but a shared memory zone (*zone*) is required.

zone

Syntax	`zone` *name* [*size*];
Default	—
Context	upstream

Defines the name and size of the shared memory zone that stores the group's configuration and runtime state, shared between worker processes. Multiple groups can use the same zone. In this case, it is sufficient to specify the size only once.

Note

The zone's content is only preserved on reload when the configured `size` is unchanged. Any size change — increase or decrease — causes the zone to be re-created empty.

Note

Upstream metrics are collected only when this zone is configured. Without it, the group does not appear in */status/stream/upstreams/<upstream>*, the *TCP/UDP Upstreams Widget*, or *Prometheus* output, and no warning is logged.

backup_switch (PRO)

Added in version 1.10.0: PRO

Syntax	`backup_switch permanent`[=*time*];
Default	—
Context	upstream

The directive enables the ability to start server selection not from the primary group, but from the *active* group, i.e., the one where a server was successfully found previously. If a server cannot be found in the active group for the next request, and the search moves to the backup group, this backup group becomes active, and subsequent requests are first directed to servers in this group.

If the `permanent` parameter is defined without a *time* value, the group remains active after selection, and automatic re-checking of groups with lower priority levels does not occur. If *time* is specified, the active status of the group expires after the specified interval, and the load balancer again checks groups with lower priority levels, returning to them if the servers are working normally.

Example:

```
upstream media_backend {
    zone media_backend 1m;
    server primary1.example.com:1935;
    server primary2.example.com:1935;

    server reserve1.example.com:1935 backup;
    server reserve2.example.com:1935 backup;

    backup_switch permanent=2m;
}
```

If the load balancer switches from primary servers to the backup group, all subsequent requests are handled by this backup group for 2 minutes. After 2 minutes expire, the load balancer re-checks the primary servers and makes them active again if they are working normally.

feedback (PRO)

Syntax	`feedback` *variable* [`inverse`] [`factor=`*number*] [`account=`*condition_ variable*];
Default	—
Context	upstream

Enables a feedback-based load balancing mechanism for the `upstream`. It dynamically adjusts load balancing decisions by multiplying each proxied server's weight by the average feedback value, which changes over time based on the *variable* value and is subject to an optional condition.

The following parameters can be specified:

`variable`	The variable from which the feedback value is taken. It should represent a performance or health metric; it is assumed to be provided by the server. The value is evaluated with each response from the server and factored into the moving average according to `inverse` and `factor` settings.
`inverse`	If the parameter is set, the feedback value is interpreted inversely: lower values indicate better performance.
`factor`	The factor by which the feedback value is weighted when calculating the average. Valid values are integers from 0 to 99. Default is `90`. The average is calculated using the exponential smoothing formula. The larger the factor, the less new values affect the average; if `90` is specified, 90% of the previous value will be taken and only 10% of the new value.
`account`	Specifies a condition variable that controls how connections are accounted for in the calculation. The average value is updated with the feedback value only if the condition variable is not equal to `""` or `"0"`. **Note** By default, traffic from *probes* is not included in the calculation; combining the *$upstream_probe* variable with `account` allows including them or even excluding 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 configuration categorizes servers by feedback levels based on protocols used in individual sessions, and also adds a condition on *$upstream_probe* to account only for `high_priority` probes or regular client sessions.

hash

Syntax	`hash` *key* [`consistent`];
Default	—
Context	upstream

Specifies a load balancing method for the group where client-server mapping is determined using a hashed key value. The key can contain text, variables, and their combinations. Note that any addition or removal of servers from the group may result in remapping of most keys to different servers. The method is compatible with the Perl Cache::Memcached library.

Usage example:

```
hash $remote_addr;
```

When using domain names that resolve to multiple IP addresses (for example, with the `resolve` parameter), the server does not sort the received addresses, so their order may differ across different servers, which affects client distribution. To ensure consistent distribution, use the `consistent` parameter.

If the `consistent` parameter is specified, the ketama consistent hashing method will be used instead of the above method. The method ensures that when a server is added to or removed from the group, only a minimal number of keys will be remapped to other servers. Using the method for caching servers provides a higher cache hit ratio. The method is compatible with the Perl Cache::Memcached::Fast library with the `ketama_points` parameter set to 160.

least_conn

Syntax	`least_conn`;
Default	—
Context	upstream

Specifies a load balancing method for the group where a connection is passed to the server with the least number of active connections, taking into account server weights. If several servers are suitable, they are selected cyclically (round-robin) with their weights taken into account.

least_time (PRO)

Syntax	`least_time` `connect` | `first_byte` | `last_byte` [`factor=`*number*] [`account=`*condition_variable*];
Default	—
Context	upstream

Specifies a load balancing method for the group where the probability of passing a connection to an active server is inversely proportional to its average response time; the lower it is, the more connections the server will receive.

`connect`	The directive takes into account the average time to establish a 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 full response.

`factor`	Performs the same function as *response_time_factor (PRO)*, and overrides it if the parameter is specified.
`account`	Specifies a condition variable that controls which connections are accounted for in the calculation. The average value is updated only if the connection's condition variable is not equal to `""` or `"0"`. **Note** By default, *probes* are not included in the calculation; combining the *$upstream_probe* variable with `account` allows including them or even excluding everything else.

Current values are presented as `connect_time`, `first_byte_time`, and `last_byte_time` in the server's `health` object among *upstream metrics* in the API.

random

Syntax	`random` [`two`];
Default	—
Context	upstream

Specifies a load balancing method for the group where a connection is passed to a randomly selected server, taking into account server weights.

If the optional `two` parameter is specified, Angie randomly selects two servers and then chooses a server using the specified method. The default method is *least_conn*, which passes the connection to the server with the least number of active connections.

response_time_factor (PRO)

Syntax	`response_time_factor` *number*;
Default	`response_time_factor 90;`
Context	upstream

Specifies for the *least_time (PRO)* load balancing method the smoothing factor for the **previous** value when calculating the average response time using the exponentially weighted moving average formula.

The larger the specified *number*, the less new values affect the average; if `90` is specified, 90% of the previous value will be taken and only 10% of the new value. Valid values are from 0 to 99 inclusive.

Current calculation results are presented as `connect_time` (time to establish a connection), `first_byte_time` (time to receive the first byte of the response), and `last_byte_time` (time to receive the full response) in the server's `health` object among *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

Changed in version 1.10.0: PRO

Changed in version 1.11.0: PRO

Syntax	`sticky route` *value...*; `sticky learn zone=`*zone* `create=`*$create_var1...* `lookup=`*$lookup_var1...* [`connect`] [`norefresh`] [`timeout=`*time*]; `sticky learn lookup=`*$lookup_var1...* `remote_action=`*uri* `remote_result=`*$remote_var* [`remote_uri=`*uri*];
Default	—
Context	upstream

Configures sticky sessions between clients and upstream servers, depending on the mode specified in the first parameter. To gradually take servers with `sticky` out of rotation, you can use the `drain` option (PRO) in the *server* block.

Warning

The `sticky` directive must appear after all load balancing method directives, otherwise it won't work.

`route` mode

This mode uses predefined route identifiers that can be embedded in connection properties accessible to Angie. It is less flexible as it depends on predefined values, but is better suited if such identifiers are already in use.

Here, when establishing a connection, the upstream server can assign a route to the client and return its identifier in a way known to both parties. The route identifier should use the value of the *sid* parameter of the *server* directive. Note that the parameter is additionally hashed if the *sticky_secret* directive is specified.

Subsequent connections from clients wishing to use this route must contain the identifier issued by the server, in such a way that it ends up in Angie variables.

The directive parameters specify strings that may include variables for routing. To select the server where the incoming connection is directed, the first non-empty value is used; it is then compared with the *sid* parameter of the *server* directive. If server selection fails or the selected server cannot accept the connection, another server will be selected according to the configured load balancing method.

Here Angie looks for the route identifier in the `$route` variable, which receives its value based on *$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 {

        zone backend 1m;

        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)

In this mode, a dynamically generated key is used to bind a client to a specific upstream server; this mode is more flexible as it assigns servers on the fly, stores sessions in a shared memory zone, and supports various ways of passing session identifiers.

Here, a session is created based on connection properties coming from the upstream server. The `create` and `lookup` parameters list variables that specify how new sessions are created and existing ones are found. Both parameters can be used multiple times.

The session identifier is the value of the first non-empty variable specified with `create`; for example, this could be the *upstream server name*.

Sessions are stored in a shared memory zone; its name and size are specified by the `zone` parameter. If a session has not been accessed within the *time* specified by `timeout`, it is deleted. The default value is 1 hour.

By default, Angie extends the session lifetime by updating the last access timestamp each time it is used. The `norefresh` parameter changes this behavior: the session expires strictly by timeout, even if it is being used.

Subsequent connections from clients wishing to use a session must contain its identifier. The `lookup` parameter searches for the session identifier in the connection using the specified list of variables, stopping at the first non-empty one. If nothing is found, the request is considered new. The value of the found identifier is matched against sessions in shared memory. If server selection fails or the selected server cannot handle the connection, another server will be selected according to the configured load balancing method.

The `connect` parameter allows creating a session immediately after establishing a connection with the upstream server. Without it, the session is created only after connection processing is complete. (In the case of UDP connections, sessions are created immediately after server selection.)

In the example, Angie creates and looks up sessions using the *$rdp_cookie* variable:

```
stream {

    upstream backend {

        zone backend 1m;

        server 127.0.0.1:3390;
        server 127.0.0.1:3391;

        sticky learn lookup=$rdp_cookie create=$rdp_cookie zone=sessions:1m;
    }

    server {
```

```
        listen 127.0.0.1:3389;

        rdp_preread on;

        proxy_pass backend;
    }
}
```

`learn` mode with `remote_action` (PRO 1.10.0+)

The `remote_action` and `remote_result` parameters allow dynamically assigning and managing session identifiers using a remote session store (PRO).

Unlike `learn` mode with `zone`, this mode does not cache sessions locally and queries the remote store for each connection.

The `remote_action` parameter must point to a `location` in the *client* context. The `remote_uri` parameter specifies the URI of the client HTTP request to the specified `location`. By default, it is `/`. The `remote_uri` value can contain variables.

The general operating principle of this mode is as follows: if a session identifier is not found locally, Angie sends a synchronous subrequest to a remote store specified by the `remote_action` parameter.

When a new connection arrives, Angie performs the following actions:

- First, the session identifier is extracted from the first non-empty variable in the `lookup` list. If all variables are empty, the normal load balancing algorithm is used without sticky sessions.
- Then Angie sends a synchronous HTTP subrequest to the remote store specified by the `remote_action` parameter, which should contain in a format understood by the store:
  - the *session* identifier from the `lookup` parameter (in the configuration, this is the *$sticky_sessid* variable);
  - the identifier of the preselected *server*: the value of the `sid=` parameter from the `server` directive, if specified, or the MD5 hash of the server name (in the configuration, this is the *$sticky_sid* variable).

  The `$sticky_sessid` and `$sticky_sid` variables are automatically exported to the HTTP context with the `stream_` prefix: `$stream_sticky_sessid`, `$stream_sticky_sid`. This allows using them directly in HTTP directives, for example via HTTP headers using *proxy_set_header*.
- The remote store processes the request and returns an HTTP response:

  A response with code 200, 201, or 204 confirms the selected server. The remote store can simultaneously return an alternative server identifier in an HTTP header or in the response body (PRO); it can be extracted via `remote_result`.

  When receiving any other HTTP code from the store (including network errors and timeouts) or a non-existent server identifier, Angie uses the originally selected server.

The server identifier is extracted from the remote store response via the `remote_result` parameter: it can specify variables with the `upstream_http_` prefix, which are created automatically by Angie to access HTTP response headers from the remote store, or *$sent_body* to use the response body. For example, the `X-Sid: server1` header in such a response becomes accessible in the `$upstream_http_x_sid` variable with the value `server1`.

Below is a simplified configuration example. The remote store returns the server identifier in the `X-Sticky-Sid` header and thus confirms or overrides Angie's selection:

```
http {

    client {

        location @sticky_client1 {
```

```
            # use variables from the stream upstream;
            # it adds these variables to the HTTP context with the stream_* prefix
            proxy_set_header X-Sticky-Sessid $stream_sticky_sessid;
            proxy_set_header X-Sticky-Sid $stream_sticky_sid;
            proxy_set_header X-Sticky-Last $msec;
            proxy_pass http://127.0.0.1:8080;

            proxy_cache remote;
            proxy_cache_valid 200 1d;
            proxy_cache_key $scheme$proxy_host$request_uri$stream_sticky_sessid;
        }
    }
}

stream {

    upstream u {

        zone u 1m;

        server 127.0.0.1:8081 sid=backend-01;
        server 127.0.0.1:8082 sid=backend-02;

        sticky learn lookup=$remote_addr             # stream variable
        remote_action=@sticky_client1                # location from client block
        remote_result=$upstream_http_x_sticky_sid    # HTTP variable
        remote_uri=/foo;                             # default is /
    }

    server {

        listen 127.0.0.1:8080;
        proxy_pass u;
    }
}
```

Here, with the following response from the remote store:

```
HTTP/1.1 200 OK
...
X-Sticky-Sid: backend-01
X-Session-Info: active
```

Two variables become available:

- `$upstream_http_x_sticky_sid`, with the value `backend-01`;
- `$upstream_http_x_session_info`, with the value `active`.

Since the `$upstream_http_x_sticky_sid` variable is specified in the `remote_result` parameter, its value will be used to select the server with `sid=backend-01`.

The `sticky` directive takes into account the state of servers in *upstream*:

- Servers marked as `down` or temporarily unavailable due to failures are excluded from selection.
- Servers that have reached the maximum number of connections (when using `max_conns`) are temporarily skipped.
- Servers with the `drain` option (PRO) can be selected for creating new sessions in `sticky` mode

when identifiers match.

- If a previously unavailable server recovers, `sticky` automatically resumes using it.

The behavior of `sticky` can be further configured with the *sticky_secret* and *sticky_strict* directives. If `sticky` fails to select a server or it is unavailable, the request will be handled according to the selected load balancing method, unless the `sticky_strict` directive is enabled. In `sticky_strict on;` mode, the request is rejected with an error.

Shared memory zones specified in the `zone` parameter of the `sticky` directive cannot be shared between different `upstream` groups; each group must use its own zone.

sticky_secret

Syntax	`sticky_secret` *string*;
Default	—
Context	upstream

Adds *string* as salt to the MD5 hash function for the *sticky* directive in `route` mode. *String* can contain variables, for example `$remote_addr`:

```
upstream backend {
    zone backend 1m;
    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;
}
```

The salt is added after the hashed value; to independently verify the hashing mechanism:

```
$ echo -n "<VALUE><SALT>" | md5sum
```

sticky_strict

Syntax	`sticky_strict on` | `off`;
Default	`sticky_strict off;`
Context	upstream

When enabled, Angie will return a connection error to the client if the desired server is unavailable, instead of falling back to another available server, which is the default behavior when no matching server is found.

Built-in Variables

The `stream_upstream` module supports the following built-in variables:

`$sticky_sessid`

Used with `remote_action` in *sticky*; stores the initial session identifier taken from `lookup`.

`$sticky_sid`

Used with `remote_action` in *sticky*; stores the server identifier previously associated with the session.

`sticky_sid` contains the value of the `sid=` parameter from the `server` directive in the *upstream* block, if specified, or the MD5 hash of the server name.

`$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 proxying, 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 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_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 and colons 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`

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 a server group where sticky is not enabled.
`NEW`	Connection does not contain sticky information.
`HIT`	Connection with sticky information routed to the desired server.
`MISS`	Connection with sticky information routed to a server 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 (`\r\n`) rather than just `\n`.

Directives

upstream_probe (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 servers within the *upstream* group specified in the *proxy_pass* directive in the same `server` context where the `upstream_probe` directive is located.

A server passes the probe if the request to it succeeds, considering all parameter settings of the `upstream_probe` directive and all parameters that affect how upstreams are used by the `server` context where it is defined, 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 server is being checked, so the server doesn't receive client requests until the probe is passed.
`persistent`	Setting this parameter requires enabling `essential` first; `persistent` servers 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 server unhealthy. By default — 1.
`passes`	Number of subsequent passed probes that renders the server healthy. 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 servers' health: • `always` — servers are probed regardless of their state; • `idle` — probes affect unhealthy servers and servers where `interval` has elapsed since the last client request. • `onfail` — only unhealthy servers 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 probe: inline data with the `data:` prefix or a file path (absolute or relative to `/usr/local/angie/`). When using a file: • The *worker process* opens and reads the file on each access; the content is not cached in memory. • Configuration reload is not required when the file changes; the new content will be read on the next access. • Required access permissions: `644` for the file, `755` for the directory. • Update files using the move command (`mv`), not by direct editing.

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 server won't receive client requests until it passes *all* `essential` probes configured for it, skipping `persistent` ones if the configuration was reloaded and the server was deemed healthy prior to that. If there are no such probes, the server is considered healthy.
- The server is considered unhealthy and won't receive client requests, if *any* of the probes configured for it hits `fails` or the server reaches *max_fails*.
- For an unhealthy server to be considered healthy again, *all* probes configured for it must reach their respective `passes`; after that, *max_fails* is also considered.

upstream_probe_timeout (PRO)

Syntax	`upstream_probe_timeout` *time*;
Default	`upstream_probe_timeout 50s;`
Context	server

Sets the maximum idle *time* for a connection established with the server for health probes configured using the *upstream_probe (PRO)* directive; if this limit is exceeded, the connection will be closed.

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

Changed in version 1.10.0.

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;
```

Note

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. **Warning** 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+. **Warning** Inappropriate use of this option may have its security implications.
`multipath`	enables accepting connections via Multipath TCP (MPTCP) protocol, supported in Linux kernel starting from version 5.6. This parameter is **incompatible** with `udp`.

`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 the entire header is not 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.

Note

Prefer a local trusted resolver such as `127.0.0.53` (systemd-resolved) over a public one (e.g. `8.8.8.8`). Public resolvers expose DNS queries to third parties and increase susceptibility to cache-poisoning attacks.

Note

The directive value is inherited by nested blocks and can be overridden in them if necessary. Within a single block, the directive may only be specified once. If it is repeated, the last definition takes effect.

By default, Angie caches answers using the TTL value of a response. If the `resolver` directive is not specified and no dynamic DNS queries are performed (for example, when using fixed names in *Proxy* without variables), specifying a resolver is not required: names will be resolved at startup using the system resolver. The optional `valid` parameter allows overriding this:

`valid`	*optional* parameter allows overriding the response cache 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 to use DNS servers in a properly secured trusted local network.

Tip

When running in Docker, use the corresponding 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;

error_log_user_tag

Syntax	`error_log_user_tag` *value*;
Default	—
Context	stream, server

Adds a session-specific tag to error log records. The *value* is a *complex value* and can use variables. The directive can be specified multiple times to add multiple tags. Tags can be matched with `filter=tag:` in *error_log*.

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.

Warning

In the `stream` module, the `server_name` directive is based 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 the current context in one shared memory zone:

```
server {

    listen 80;
    server_name *.example.com;

    status_zone single;
    # ...
}
```

The alternative syntax allows specifying the following 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 with the same `$server_addr` value are grouped into `host_zone`. Metrics are collected separately for each unique `$server_addr` until the number of metric groups reaches 10. After that, any new `$server_addr` values will be added to the `server_zone` group:

```
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 split between individual servers in the API output.

Note

These metrics are collected only when `status_zone` is set. Without it, the server does not appear in */status/stream/server_zones/<zone>*, the *TCP/UDP Zones Widget*, or *Prometheus* output, and no warning is logged.

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 connections and connections to proxied servers.

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 core stream module supports the following built-in 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 the client

`$bytes_sent`

number of bytes sent to the client

`$connection`

connection serial number

`$hostname`

host name

`$msec`

current time in seconds with milliseconds resolution

`$nginx_version`

nginx version

`$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 obtained from the PROXY protocol header. The *name* can be a TLV type name or its numeric value. In the latter case, the value is specified in hexadecimal and must start with *0x*:

```
$proxy_protocol_tlv_alpn
$proxy_protocol_tlv_0x01
```

SSL TLVs can also be accessed by both TLV type name and its numeric value, both must start with `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 identifier
- `netns (0x30)` - namespace name
- `ssl (0x20)` - SSL TLV structure in binary format

The following SSL TLV type names are supported:

- `ssl_version (0x21)` - SSL version used in client connection
- `ssl_cn (0x22)` - 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 supported is the following special SSL TLV type name:

- `ssl_verify` - client certificate verification result: 0 if the client presented a certificate and it was successfully verified, or 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`

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 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 a 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 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; 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/xoauth2/oauthbearer/none
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 APOP or CRAM-MD5 is used, the 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
```

When XOAUTH2 or OAUTHBEARER is used, the `Auth-User` and `Auth-Pass` headers contain the user identity and bearer token extracted from the initial SASL response.

If the `Auth-User` header exists in the response, it overrides the username used to authenticate with the backend.

For SMTP, the response additionally takes into account the `Auth-Error-Code` header — if it 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 by default.

For XOAUTH2 and OAUTHBEARER, an error response may also include the `Auth-Error-SASL` header. Its value is sent to the client as an additional SASL challenge (SMTP: *334*, IMAP/POP3: *+*). After the client responds with an empty response for XOAUTH2 or `AQ==` for OAUTHBEARER, the error from `Auth-Status` is returned.

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 SSL/TLS client connections, 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 such as listing mailboxes and retrieving messages, and provides secure authentication and message status management.

Directives

imap_auth

Changed in version 1.11.0.

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
`xoauth2`	AUTH=XOAUTH2
`oauthbearer`	AUTH=OAUTHBEARER

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 such as downloaded or deleted.

Directives

pop3_auth

Changed in version 1.11.0.

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
`xoauth2`	AUTH=XOAUTH2
`oauthbearer`	AUTH=OAUTHBEARER

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 mail servers. It establishes connections with servers, performs secure authentication using plain text, SSL/TLS, or STARTTLS, properly routes client traffic, and supports flexible authentication method 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

Determines whether to pass the error message obtained during authentication on the backend to the client.

Usually, if authentication in Angie is successful, the backend cannot return an error. If it nevertheless returns an error, it means some internal error has occurred. In such cases the backend message may contain information that should not be shown to the client. However, responding with an error for the correct password is 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 mail servers. It establishes connections to SMTP servers, supports secure authentication using LOGIN or PLAIN methods, provides STARTTLS and SSL/TLS encryption, and routes client requests based on authentication results.

Directives

smtp_auth

Changed in version 1.11.0.

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
`xoauth2`	AUTH XOAUTH2
`oauthbearer`	AUTH OAUTHBEARER
`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 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 the 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.

Note

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_compression

Syntax	`ssl_certificate_compression on | off;`
Default	`ssl_certificate_compression off;`
Context	mail, server

Enables TLS 1.3 compression of server certificates.

Note

The directive is supported when using OpenSSL 3.2 or higher; the list of supported compression algorithms is provided by the library.

Note

The directive is supported when using BoringSSL; the list of supported compression algorithms includes `zlib`.

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 `store:scheme:id` can be specified instead of the `file`, which is used to load a secret key with a specified *id* and OpenSSL provider registered URI *scheme*, such as pkcs11.

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.

Warning

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.

Note

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.

Warning

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.

Warning

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.

Note

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.

Note

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

Enables the specified protocols.

Note

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	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 use in cases when a service that is external to Angie performs the actual certificate verification. The contents of the certificate are 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 source, 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

Changed in version 1.10.0.

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 connections. It is possible to specify just the *port*, and Angie will listen on all available IPv4 interfaces (and IPv6, if enabled). 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;
```

Note

Different servers must listen on different *address:port* pairs.

`ssl`	specifies that all connections accepted on this port should work in SSL mode.
`proxy_protocol`	specifies 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`	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`	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.
`multipath`	enables accepting connections via Multipath TCP (MPTCP), supported in Linux kernel version 5.6 and later.

`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, 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.

Different servers must listen on different *address:port* pairs.

mail

Syntax	`mail { ... }`
Default	—
Context	main

Provides the configuration file context in which the mail server directives are specified.

max_commands

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*] | `off`;
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 special value `off` disables resolving of the client hostname and cancels an inherited directive value.

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.

Note

Prefer a local trusted resolver such as `127.0.0.53` (systemd-resolved) over a public one (e.g. `8.8.8.8`). Public resolvers expose DNS queries to third parties and increase susceptibility to cache-poisoning attacks.

Note

The directive value is inherited by nested blocks and can be overridden in them if necessary. Within a single block, the directive can only be specified once. If it is repeated, the last definition takes effect.

By default, Angie caches answers using the TTL value of a response. If the `resolver` directive is not specified and no dynamic DNS queries are performed (for example, when using fixed names in *Proxy* without variables), specifying a resolver is not required: names will be resolved at startup using the system resolver. The optional `valid` parameter allows overriding this:

`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 collection of information about DNS server requests and responses in the specified zone

Tip

To prevent DNS spoofing, it is recommended to configure 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 name resolution, 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 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 server performance by providing detailed information about memory usage, CPU load, and other performance metrics.

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

Note

This module requires the gperftools library.

Configuration Example

```
google_perftools_profiles /var/log/angie/perftools;
```

Profiles will be stored in files like `/var/log/angie/perftools.<worker process PID>`.

Directives

google_perftools_profiles

Syntax	`google_perftools_profiles` *file prefix*;
Default	—
Context	main

Sets the filename prefix where profiling information for the Angie worker process will be stored. The worker process ID is appended at the end of the filename after a dot, for example: `/var/log/angie/perftools.1234`.

WASM Module

WAMR

The module provides 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 provides 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 in Angie.

Note

Angie-specific APIs can be explicitly allowed 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 the specified *size*, thus limiting the maximum amount of stack space available for executing WASM code.

The core module that implements basic WASM functionality in Angie: it 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 WASM capabilities 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=`*identifier* [`fs=`*host_path*`:`*guest_path*]... [`api=`*api*]... [`type=command` | `reactor`]
Default	—
Context	wasm_modules

Loads a module from a disk *file* and assigns it a unique *identifier* (required parameter). During loading, verification occurs to ensure the module can be instantiated.

The directive supports the following parameters:

`fs`	Allows the guest to access a directory on the host. The parameter can be specified multiple times for different directories.
`api`	Explicitly restricts the list of APIs allowed for the module by listing them. If the module attempts to use unavailable APIs (not listed here), an "API not found" error is returned. By default, all APIs are available to the module.
`type`	Controls the lifecycle of the loaded module. • In `command` mode, the machine executes once and its state is destroyed after execution. • In `reactor` mode, the machine effectively runs indefinitely, allowing code to be executed multiple times. This requires careful memory management: if resources are not freed, memory leaks can occur.

wasm_modules

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

A top-level directive that provides the configuration file context in which WASM directives should be specified. It can contain commands for loading WASM modules and configuring parameters specific to a particular runtime.

Core Module

Core	Management of service files, processes, and other Angie modules.

HTTP Modules

HTTP	Core functionality for processing HTTP requests and responses, managing the HTTP server, connections, and static files.
Access	Access control based on IP addresses and CIDR ranges.
ACME	Automatic retrieval and renewal of SSL certificates using the ACME protocol for HTTP servers.

continues on next page

Table 1 – continued from previous page

Docker	Dynamic updating of proxied server groups based on Docker container labels.
Addition	Insertion of a specified snippet before or after the response body.
API	RESTful HTTP interface for obtaining basic web server information and statistics in JSON format, as well as managing groups of proxied servers.
Auth Basic	Basic HTTP authentication for access control based on 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	Serving a one-pixel transparent GIF.
FastCGI	Proxying requests to a FastCGI server.
FLV	Pseudo-streaming of Flash Video (FLV) files.
Geo	Converting IP addresses into specified variable values.
GeoIP	Obtaining IP address data based on geolocation using MaxMind GeoIP databases.
gRPC	Proxying requests to a gRPC server.
GunZIP	Decompressing GZip-compressed responses for modification and in cases where the client does not support compression.
GZip	Compressing responses using the GZip method to save traffic.
GZip Static	Serving static files pre-compressed using the GZip method.
Headers	Modifying response header fields.
HTTP2	Processing requests using the HTTP/2 protocol.
HTTP3	Processing requests using the HTTP/3 protocol.
Image Filter[1]	Image transformation.
Index	Configuration of index files that serve requests ending with a slash (`/`).
Limit Conn	Limiting the number of concurrent requests (active connections) for protection against overload.
Limit Req	Limiting request frequency for protection against overload and password guessing.
Log	Configuration of request logs for tracking resource access for monitoring and analysis purposes.
Map	Converting variables based on predefined key-value pairs.
Metric	Custom numeric metrics in the real-time statistics API.
Memcached	Retrieving responses from a Memcached server.
Mirror	Mirroring requests to other servers.
MP4	Pseudo-streaming of MP4 files.
Perl[Page 451, 1]	Handlers for extending functionality by specifying additional logic in the Perl language.
Prometheus	Server metrics in Prometheus-compatible format for monitoring and statistics collection.

continues on next page

Table 1 – continued from previous page

Proxy	Reverse proxying requests to other HTTP servers.
Random Index	Random selection of an index file for requests ending with a slash (`/`).
RealIP	Determining client address and port when operating behind another proxy server.
Referer	Validation of `Referer` header values.
Rewrite	Request URI modification, redirects, variable setting, and conditional configuration selection.
SCGI	Proxying requests to an SCGI server.
Secure Link	Creating secure links with the ability to limit access time.
Slice	Splitting requests into multiple subrequests for individual fragments for better caching of large responses.
Split Clients	Creating variables for A/B testing, canary releases, sharding, and other scenarios requiring proportional group splitting.
SSI	Processing SSI (Server Side Includes) commands in responses.
SSL	SSL/TLS configuration for processing HTTPS requests.
Stub Status (deprecated)	Global connection and request counters in text format.
Sub	Search and replace fragments in the response body.
Upstream	Configuration of proxied server groups for load balancing.
Upstream Probe	Configuration of active health probes for proxied server groups.
UserID	Issuing and processing cookies with unique client identifiers for session tracking and analytics.
uWSGI	Proxying requests to a uWSGI server.
XSLT[1]	Transforming XML documents using the XSLT language.

[1] In our builds, these modules are compiled dynamically and installed as separate packages; for details, see the description of each module.

Stream Modules

Stream	Core stream server functionality for balancing TCP and UDP protocols at the L4 level.
Access	Access control based on IP addresses and CIDR ranges.
ACME	Automatic retrieval and renewal of SSL certificates using the ACME protocol for stream servers.
Geo	Converting IP addresses into specified variable values.
GeoIP	Obtaining IP address data based on geolocation using MaxMind GeoIP databases.
Limit Conn	Limiting the number of concurrent connections for protection against overload.
Log	Configuration of session logs for tracking resource access for monitoring and analysis purposes.
Map	Converting variables based on predefined key-value pairs.
MQTT Preread	Reading client identifier and username from MQTT connections before making load balancing decisions.
Pass	Passing accepted connections directly to a configured listening socket.
Proxy	Configuration of proxying to other servers.
RDP Preread	Reading cookies from RDP connections before making load balancing decisions.
RealIP	Determining client address and port when operating behind another proxy server.
Return	Sending a specified value to the client upon connection without further proxying.
Set	Setting specified variable values.
Split Clients	Creating variables for A/B testing, canary releases, sharding, and other scenarios requiring proportional group splitting.
SSL	SSL/TLS and DTLS protocol termination.
SSL Preread	Extracting information from `ClientHello` messages without SSL/TLS termination and before making load balancing decisions.
Upstream	Configuration of proxied server groups for load balancing.
Upstream Probe	Configuration of active health probes for proxied server groups.

Mail Modules

Mail	Core mail proxy server functionality.
Auth HTTP	User authentication and server selection for subsequent proxying using HTTP requests to an external server.
IMAP	IMAP protocol support.
POP3	POP3 protocol support.
Proxy	Configuration of proxying to other servers.
RealIP	Determining client address and port when operating behind another proxy server.
SMTP	SMTP protocol support.
SSL	SSL/TLS and StartTLS protocol support.

Google PerfTools Module

Google PerfTools	Responsible for integration with the Google Performance Tools library for application profiling and performance analysis.

WASM Modules

WASM[Page 451, 1]	Core WASM functionality enabling WASM code execution in Angie.
WAMR	Integration with WebAssembly Micro Runtime.
Wasmtime	Integration with the Wasmtime runtime environment.

3.2.2 Built-in Variables

HTTP Modules	*Stream Modules*
$acme_cert_key_<name>	*$acme_cert_key_<name>*
$acme_cert_<name>	*$acme_cert_<name>*
$acme_hook_challenge	
$acme_hook_client	
$acme_hook_domain	
$acme_hook_keyauth	
$acme_hook_name	
$acme_hook_token	
$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	
$gzip_ratio	
$host	
$hostname	*$hostname*
$http2	
$http3	
$http_<name>	
$https	
$invalid_referer	
$is_args	
$is_request_port	
$limit_conn_status	*$limit_conn_status*
$limit_rate	
$limit_req_status	
$memcached_key	
$metric_<name>	
$metric_<name>_key and *$metric_<name>_value*	
$metric_<name>_key and *$metric_<name>_value*	
$metric_<name>_value_<metric>	
$modern_browser	

continues on next page

Table 2 – continued from previous page

HTTP Modules	*Stream Modules*
	$mqtt_preread_clientid
	$mqtt_preread_username
$msec	*$msec*
$msie	
Protocol	
$nginx_version	*$nginx_version*
$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_port	
$request_time	
$request_uri	
$scheme	
$secure_link	
$secure_link_expires	
$sent_http_<name>	
$sent_body	
$sent_trailer_<name>	
$server_addr	*$server_addr*
$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

continues on next page

Table 2 – continued from previous page

HTTP Modules	*Stream Modules*
$ssl_client_fingerprint	*$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*
$ssl_client_sigalg	*$ssl_client_sigalg*
$ssl_client_v_end	*$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*
$ssl_encrypted_hello	*$ssl_encrypted_hello*
	$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*
$ssl_sigalg	*$ssl_sigalg*
$status	*$status*
$sticky_sessid	*$sticky_sessid*
$sticky_sid	*$sticky_sid*
$time_iso8601	*$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_cache_key	
$upstream_connect_time	*$upstream_connect_time*
$upstream_cookie_<name>	
	$upstream_first_byte_time
$upstream_header_time	
$upstream_http_<name>	
$upstream_request_method	
$upstream_probe (PRO)	*$upstream_probe (PRO)*
$upstream_probe_body (PRO)	
	$upstream_probe_response (PRO)
$upstream_response_length	
$upstream_response_time	
	$upstream_session_time
$upstream_status	
$upstream_sticky_status	*$upstream_sticky_status*
$upstream_trailer_<name>	

continues on next page

Table 2 – continued from previous page

HTTP Modules	*Stream Modules*
$upstream_queue_time	
$uri	

3.2.3 NJS API Reference

The NJS module provides objects, methods, and properties for extending Angie functionality.

This reference contains only NJS-specific properties, methods, and modules not compliant with ECMAScript. Definitions of NJS properties and methods compliant with ECMAScript can be found in the ECMAScript specification.

Angie Objects

HTTP Request

- `r.args{}`
- `r.done()`
- `r.error()`
- `r.finish()`
- `r.headersIn{}`
- `r.headersOut{}`
- `r.httpVersion`
- `r.internal`
- `r.internalRedirect()`
- `r.log()`
- `r.method`
- `r.parent`
- `r.remoteAddress`
- `r.requestBody`
- `r.requestBuffer`
- `r.requestText`
- `r.rawHeadersIn[]`
- `r.rawHeadersOut[]`
- `r.responseBody`
- `r.responseBuffer`
- `r.responseText`
- `r.return()`
- `r.send()`
- `r.sendBuffer()`
- `r.sendHeader()`
- `r.setReturnValue()`
- `r.status`
- `r.subrequest()`

- `r.uri`
- `r.rawVariables{}`
- `r.variables{}`
- `r.warn()`

The HTTP request object is available only in the HTTP JS module. Before 0.8.5, all string properties of the object were byte strings.

`r.args{}`

Request arguments object, read-only.

The query string is returned as an object. Since 0.7.6, duplicate keys are returned as an array, keys are case-sensitive, both keys and values are percent-decoded.

For example, the query string

```
a=1&b=%32&A=3&b=4&B=two%20words
```

is converted to `r.args` as:

```
{a: "1", b: ["2", "4"], A: "3", B: "two words"}
```

More advanced parsing scenarios can be achieved with the *Query String* module and the `$args` variable, for example:

```
import qs from 'querystring';

function args(r) {
    return qs.parse(r.variables.args);
}
```

The arguments object is evaluated at the first access to `r.args`. If only a single argument is needed, for example `foo`, Angie variables can be used:

```
r.variables.arg_foo
```

Here, the Angie variables object returns the first value for a given key, case-insensitive, without percent-decoding.

To convert `r.args` back to a string, the Query String `stringify` method can be used.

`r.done()`
: After calling this function, the next data chunks will be passed to the client without calling `js_body_filter` (0.5.2). May be called only from the `js_body_filter` function.

`r.error(string)`
: Writes a `string` to the error log on the `error` level of logging.

> **Note**
>
> As Angie has a hardcoded maximum line length limit, only the first 2048 bytes of the string can be logged.

`r.finish()`
: Finishes sending a response to the client.

`r.headersIn{}`
: Incoming headers object, read-only.

The `Foo` request header can be accessed with the syntax: `headersIn.foo` or `headersIn['Foo']`.

The `Authorization`, `Content-Length`, `Content-Range`, `Content-Type`, `ETag`, `Expect`, `From`, `Host`, `If-Match`, `If-Modified-Since`, `If-None-Match`, `If-Range`, `If-Unmodified-Since`, `Max-Forwards`, `Proxy-Authorization`, `Referer`, `Transfer-Encoding`, and `User-Agent` request headers can have only one field value (0.4.1). Duplicate field values in `Cookie` headers are separated by semicolon (`;`). Duplicate field values in all other request headers are separated by commas.

`r.headersOut{}`
: Outgoing headers object for the main request, writable.

  If `r.headersOut{}` is the response object of a subrequest, it represents response headers. In this case, field values in `Accept-Ranges`, `Connection`, `Content-Disposition`, `Content-Encoding`, `Content-Length`, `Content-Range`, `Date`, `Keep-Alive`, `Server`, `Transfer-Encoding`, `X-Accel-*` response headers may be omitted.

  The `Foo` response header can be accessed with the syntax: `headersOut.foo` or `headersOut['Foo']`.

  Outgoing headers should be set before a response header is sent to a client; otherwise, the header update will be ignored. This means that `r.headersOut{}` is effectively writable in:

  - the `js_content` handler before `r.sendHeader()` or `r.return()` are called
  - the `js_header_filter` handler

  Field values of multi-value response headers (0.4.0) can be set with the syntax:

```
r.headersOut['Foo'] = ['a', 'b']
```

  where the output will be:

```
Foo: a
Foo: b
```

  All previous field values of the `Foo` response header will be deleted.

  For standard response headers that accept only a single field value such as `Content-Type`, only the last element of the array will take effect. Field values of the `Set-Cookie` response header are always returned as an array. Duplicate field values in `Age`, `Content-Encoding`, `Content-Length`, `Content-Type`, `ETag`, `Expires`, `Last-Modified`, `Location`, `Retry-After` response headers are ignored. Duplicate field values in all other response headers are separated by commas.

`r.httpVersion`
: HTTP version, read-only.

`r.internal`
: Boolean value, `true` for internal locations.

`r.internalRedirect(uri)`
: Performs an internal redirect to the specified `uri`. If the URI starts with the `@` prefix, it is considered a named location. In a new location, all request processing is repeated starting from `NGX_HTTP_SERVER_REWRITE_PHASE` for ordinary locations and from `NGX_HTTP_REWRITE_PHASE` for named locations. As a result, a redirect to a named location does not check the `client_max_body_size` limit. Redirected requests become internal and can access internal locations. The actual redirect happens after the handler execution is completed.

  Note

  After redirect, a new NJS VM is started in the target location, and the VM in the original location is stopped. Values of Angie variables are kept and can be used to pass information to the target location. Since 0.5.3, a variable declared with the `js_var` directive for HTTP or Stream can be used.

> **Note**
>
> Since 0.7.4, the method accepts escaped URIs.

`r.log(string)`
: Writes a `string` to the error log on the `info` level of logging.

> **Note**
>
> As Angie has a hardcoded maximum line length limit, only the first 2048 bytes of the string can be logged.

`r.method`
: HTTP method, read-only.

`r.parent`
: References the parent request object.

`r.remoteAddress`
: Client address, read-only.

`r.requestBody`
: The property was made obsolete in 0.5.0 and was removed in 0.8.0. The `r.requestBuffer` or `r.requestText` property should be used instead.

`r.requestBuffer`
: Client request body if it has not been written to a temporary file (since 0.5.0). 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`. The property is available only in the `js_content` directive.

`r.requestText`
: The same as `r.requestBuffer`, but returns a `string`. Note that it may convert bytes invalid in UTF-8 encoding into the replacement character.

`r.rawHeadersIn[]`
: Returns an array of key-value pairs exactly as they were received from the client (0.4.1).

  For example, with the following request headers:

```
Host: localhost
Foo:  bar
foo:  bar2
```

  the output of `r.rawHeadersIn` will be:

```
[
    ['Host', 'localhost'],
    ['Foo', 'bar'],
    ['foo', 'bar2']
]
```

  All `foo` headers can be collected with the syntax:

```
r.rawHeadersIn.filter(v=>v[0].toLowerCase() == 'foo').map(v=>v[1])
```

  the output will be:

```
['bar', 'bar2']
```

  Header field names are not converted to lower case, duplicate field values are not merged.

`r.rawHeadersOut[]`
: Returns an array of key-value pairs of response headers (0.4.1). Header field names are not converted to lower case, duplicate field values are not merged.

`r.responseBody`
: The property was deprecated in 0.5.0 and was removed in 0.8.0. The `r.responseBuffer` or `r.responseText` property should be used instead.

`r.responseBuffer`
: Contains the subrequest response body, read-only (since 0.5.0). The size of `r.responseBuffer` is limited by the `subrequest_output_buffer_size` directive.

`r.responseText`
: The same as `r.responseBuffer` but returns a string (since 0.5.0). Note that it may convert bytes invalid in UTF-8 encoding into the replacement character.

`r.return(status[, string | Buffer])`
: Sends the entire response with the specified `status` to the client. The response can be a string or Buffer (0.5.0).

  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) as the second argument.

`r.send(string | Buffer)`
: Sends a part of the response body to the client. The data sent can be a string or Buffer (0.5.0).

`r.sendBuffer(data[, options])`
: Adds data to the chain of data chunks to be forwarded to the next body filter (0.5.2). The actual forwarding happens later, when all the data chunks of the current chain are processed.

  The data can be a string or Buffer. The `options` is an object used to override Angie buffer flags derived from an incoming data chunk buffer. The flags can be overridden with the following flags:

  `last`
  : Boolean, `true` if the buffer is the last buffer.

  `flush`
  : Boolean, `true` if the buffer should have the `flush` flag.

  The method may be called only from the `js_body_filter` function.

`r.sendHeader()`
: Sends the HTTP headers to the client.

`r.setReturnValue(value)`
: Sets the return value of the `js_set` handler (0.7.0). Unlike an ordinary return statement, this method should be used when the handler is a JS async function. For example:

```
async function js_set(r) {
    const digest = await crypto.subtle.digest('SHA-256', r.headersIn.host);
    r.setReturnValue(digest);
}
```

`r.status`
: Status, writable.

`r.subrequest(uri[, options[, callback]])`
: Creates a subrequest with the given `uri` and `options`, and installs an optional completion `callback`.

  A subrequest shares its input headers with the client request. To send headers different from original headers to a proxied server, the `proxy_set_header` directive can be used. To send a completely new set of headers to a proxied server, the `proxy_pass_request_headers` directive can be used.

If `options` is a string, then it holds the subrequest arguments string. Otherwise, `options` is expected to be an object with the following keys:

`args`
: Arguments string, by default an empty string is used.

`body`
: Request body, by default the request body of the parent request object is used.

`method`
: HTTP method, by default the `GET` method is used.

`detached`
: Boolean flag (0.3.9); if `true`, the created subrequest is a detached subrequest. Responses to detached subrequests are ignored. Unlike ordinary subrequests, a detached subrequest can be created inside a variable handler. The `detached` flag and callback argument are mutually exclusive.

The completion `callback` receives a subrequest response object with methods and properties identical to the parent request object.

Since 0.3.8, if a `callback` is not provided, the `Promise` object that resolves to the subrequest response object is returned.

For example, to view all response headers in the subrequest:

```
async function handler(r) {
    const reply = await r.subrequest('/path');

    for (const h in reply.headersOut) {
        r.log(`${h}: ${reply.headersOut[h]}`);
    }

    r.return(200);
}
```

`r.uri`
: Current URI in request, normalized, read-only.

`r.rawVariables{}`
: Angie variables as Buffers, writable (since 0.5.0).

`r.variables{}`
: Angie variables object, writable (since 0.2.8).

For example, to get the `$foo` variable, one of the following syntax can be used:

```
r.variables['foo']
r.variables.foo
```

Since 0.8.6, regular expression captures can be accessed using the following syntax:

```
r.variables['1']
r.variables[1]
```

Angie treats variables referenced in `angie.conf` and unreferenced variables differently. When a variable is referenced, it may be cacheable, but when it is unreferenced, it is always uncacheable. For example, when the `$request_id` variable is only accessed from NJS, it has a new value every time it is evaluated. But, when the `$request_id` is referenced, for example:

```
proxy_set_header X-Request-Id $request_id;
```

the `r.variables.request_id` returns the same value every time.

A variable is writable if:

- it was created using the `js_var` directive for HTTP or Stream (since 0.5.3)
- it is referenced in Angie configuration file

Even so, some embedded variables still cannot be assigned a value (for example, `$http_`).

`r.warn(string)`
: Writes a `string` to the error log on the `warning` level of logging.

> **Note**
>
> As Angie has a hardcoded maximum line length limit, only the first 2048 bytes of the string can be logged.

Stream Session

- `s.allow()`
- `s.decline()`
- `s.deny()`
- `s.done()`
- `s.error()`
- `s.log()`
- `s.off()`
- `s.on()`
- `s.remoteAddress`
- `s.rawVariables{}`
- `s.send()`
- `s.sendDownstream()`
- `s.sendUpstream()`
- `s.status`
- `s.setReturnValue()`
- `s.variables{}`
- `s.warn()`

The stream session object is available only in the Stream JS module. Before 0.8.5, all string properties of the object were byte strings.

`s.allow()`
: An alias to `s.done(0)` (0.2.4).

`s.decline()`
: An alias to `s.done(-5)` (0.2.4).

`s.deny()`
: An alias to `s.done(403)` (0.2.4).

`s.done([code])`
: Sets an exit `code` for the current phase handler to a code value, by default `0`. The actual finalization happens when the js handler is completed and all pending events, for example, from `ngx.fetch()` or `setTimeout()`, are processed (0.2.4).

  Possible code values:

  - `0` — successful finalization, passing control to the next phase

- `-5` — undecided, passing control to the next handler of the current phase (if any)
- `403` — access is forbidden

May be called only from a phase handler function: `js_access` or `js_preread`.

`s.error(string)`
: Writes a sent `string` to the error log on the `error` level of logging.

> **Note**
>
> As Angie has a hardcoded maximum line length limit, only the first 2048 bytes of the string can be logged.

`s.log(string)`
: Writes a sent `string` to the error log on the `info` level of logging.

> **Note**
>
> As Angie has a hardcoded maximum line length limit, only the first 2048 bytes of the string can be logged.

`s.off(eventName)`
: Unregisters the callback set by the `s.on()` method (0.2.4).

`s.on(event, callback)`
: Registers a `callback` for the specified `event` (0.2.4).

  An `event` may be one of the following strings:

  `upload`
  : New data (string) from a client.

  `download`
  : New data (string) to a client.

  `upstream`
  : New data (Buffer) from a client (since 0.5.0).

  `downstream`
  : New data (Buffer) to a client (since 0.5.0).

  The completion callback has the following prototype: `callback(data, flags)`, where `data` is string or Buffer (depending on the event type); `flags` is an object with the following properties:

  `last`
  : A boolean value, `true` if data is a last buffer.

`s.remoteAddress`
: Client address, read-only.

`s.rawVariables`
: Angie variables as Buffers, writable (since 0.5.0).

`s.send(data[, options])`
: Adds data to the chain of data chunks that will be forwarded in the forward direction: in download callback to a client; in upload to an upstream server (0.2.4). The actual forwarding happens later, when all the data chunks of the current chain are processed.

  The data can be a string or Buffer (0.5.0). The `options` is an object used to override Angie buffer flags derived from an incoming data chunk buffer. The flags can be overridden with the following flags:

`last`
: Boolean, `true` if the buffer is the last buffer.

`flush`
: Boolean, `true` if the buffer should have the `flush` flag.

The method can be called multiple times per callback invocation.

`s.sendDownstream()`
: Identical to `s.send()`, except for it always sends data to a client (since 0.7.8).

`s.sendUpstream()`
: Identical to `s.send()`, except for it always sends data from a client (since 0.7.8).

`s.status`
: Session status code, an alias to the `$status` variable, read-only (since 0.5.2).

`s.setReturnValue(value)`
: Sets the return value of the `js_set` handler (0.7.0). Unlike an ordinary return statement, this method should be used when the handler is a JS async function. For example:

```
async function js_set(r) {
    const digest = await crypto.subtle.digest('SHA-256', r.headersIn.host);
    r.setReturnValue(digest);
}
```

`s.variables{}`
: Angie variables object, writable (since 0.2.8). A variable can be writable only if it is referenced in the Angie configuration file. Even so, some embedded variables still cannot be assigned a value.

`s.warn(string)`
: Writes the sent `string` to the error log on the `warning` logging level.

> **Note**
>
> As Angie has a hardcoded maximum line length limit, only the first 2048 bytes of the string can be logged.

Periodic Session

- `PeriodicSession.rawVariables{}`
- `PeriodicSession.variables{}`

The `Periodic Session` object is provided as the first argument for the `js_periodic` handler for HTTP and Stream (since 0.8.1).

`PeriodicSession.rawVariables{}`
: Angie variables as Buffers, writable.

`PeriodicSession.variables{}`
: Angie variables object, writable.

Headers

- `Headers()`
- `Headers.append()`
- `Headers.delete()`
- `Headers.get()`
- `Headers.getAll()`

- `Headers.forEach()`
- `Headers.has()`
- `Headers.set()`

The `Headers` interface of the Fetch API is available since 0.5.1.

A new `Headers` object can be created using the `Headers()` constructor (since 0.7.10):

`Headers([init])`

`init`
: An object containing HTTP headers for prepopulating the `Headers` object, can be a `string`, an `array` of name-value pairs, or an existing `Headers` object.

A new `Headers` object can be created with the following properties and methods:

`append()`
: Appends a new value into an existing header in the `Headers` object, or adds the header if it does not already exist (since 0.7.10).

`delete()`
: Deletes a header from the `Headers` object (since 0.7.10).

`get()`
: Returns a string containing the values of all headers with the specified name separated by a comma and a space.

`getAll(name)`
: Returns an array containing the values of all headers with the specified name.

`forEach()`
: Executes a provided function once for each key/value pair in the `Headers` object (since 0.7.10).

`has()`
: Returns a boolean value indicating whether a header with the specified name exists.

`set()`
: Sets a new value for an existing header inside the `Headers` object, or adds the header if it does not already exist (since 0.7.10).

Request

- `Request()`
- `Request.arrayBuffer()`
- `Request.bodyUsed`
- `Request.cache`
- `Request.credentials`
- `Request.headers`
- `Request.json()`
- `Request.method`
- `Request.mode`
- `Request.text()`
- `Request.url`

The `Request` interface of the Fetch API is available since 0.7.10.

A new `Request` object can be created using the `Request()` constructor:

`Request[resource[, options]])`
: Creates a `Request` object to fetch that can be passed later to `ngx.fetch()`. The `resource` can be a URL or an existing `Request` object. The `options` is an optional argument that is expected to be an object with the following keys:

  `body`
  : The request body, by default is empty.

  `headers`
  : The response headers object — the object containing HTTP headers for prepopulating the `Headers` object, can be a `string`, an `array` of name-value pairs, or an existing `Headers` object.

  `method`
  : The HTTP method, by default the GET method is used.

A new `Request` object can be created with the following properties and methods:

`arrayBuffer()`
: Returns a `Promise` that resolves with an `ArrayBuffer`.

`bodyUsed`
: A boolean value, `true` if the body was used in the request.

`cache`
: Contains the cache mode of the request.

`credentials`
: Contains the credentials of the request, by default is `same-origin`.

`headers`
: The `Headers` read-only object associated with the `Request`.

`json()`
: Returns a `Promise` that resolves with the result of parsing the request body as JSON.

`method`
: Contains the request method.

`mode`
: Contains the mode of the request.

`text()`
: Returns a `Promise` that resolves with a string representation of the request body.

`url`
: Contains the URL of the request.

Response

- `Response()`
- `Response.arrayBuffer()`
- `Response.bodyUsed`
- `Response.headers`
- `Response.json()`
- `Response.ok`
- `Response.redirected`
- `Response.status`
- `Response.statusText`
- `Response.text()`
- `Response.type`

- `Response.url`

The `Response` interface is available since 0.5.1.

A new `Response` object can be created using the `Response()` constructor (since 0.7.10):

`Response[body[, options]])`
: Creates a `Response` object. The `body` is an optional argument, can be a `string` or a `buffer`, by default is `null`. The `options` is an optional argument that is expected to be an object with the following keys:

  `headers`
  : The response headers object — the object containing HTTP headers for prepopulating the `Headers` object, can be a `string`, an `array` of name-value pairs, or an existing `Headers` object.

  `status`
  : The status code of the response.

  `statusText`
  : The status message corresponding to the status code.

A new `Response()` object can be created with the following properties and methods:

`arrayBuffer()`
: Takes a `Response` stream and reads it to completion. Returns a `Promise` that resolves with an `ArrayBuffer`.

`bodyUsed`
: A boolean value, `true` if the body was read.

`headers`
: The `Headers` read-only object associated with the `Response`.

`json()`
: Takes a `Response` stream and reads it to completion. Returns a `Promise` that resolves with the result of parsing the body text as JSON.

`ok`
: A boolean value, `true` if the response was successful (status codes between 200–299).

`redirected`
: A boolean value, `true` if the response is the result of a redirect.

`status`
: The status code of the response.

`statusText`
: The status message corresponding to the status code.

`text()`
: Takes a `Response` stream and reads it to completion. Returns a `Promise` that resolves with a string.

`type`
: The type of the response.

`url`
: The URL of the response.

ngx

- `ngx.build`
- `ngx.conf_file_path`
- `ngx.conf_prefix`
- `ngx.error_log_path`

- `ngx.fetch()`
- `ngx.log()`
- `ngx.prefix`
- `ngx.version`
- `ngx.version_number`
- `ngx.worker_id`

The `ngx` global object is available since 0.5.0.

`ngx.build`
: A string containing an optional Angie build name, corresponds to the `--build=name` argument of the configure script, by default is `""` (0.8.0).

`ngx.conf_file_path`
: A string containing the file path to current Angie configuration file (0.8.0).

`ngx.conf_prefix`
: A string containing the file path to Angie configuration prefix — the directory where Angie is currently looking for configuration (0.7.8).

`ngx.error_log_path`
: A string containing the file path to the current error log file (0.8.0).

`ngx.fetch(resource, [options])`
: Makes a request to fetch a `resource` (0.5.1), which can be a URL or the `Request` object (0.7.10). Returns a `Promise` that resolves with the `Response` object. Since 0.7.0, the `https://` scheme is supported; redirects are not handled.

  If the URL in the `resource` is specified as a domain name, it is resolved using a resolver. If the `https://` scheme is specified, the `js_fetch_trusted_certificate` directive should be configured for the authentication of the `resource`'s HTTPS server.

  The `options` parameter is expected to be an object with the following keys:

  `body`
  : Request body, by default is empty.

  `buffer_size`
  : The buffer size for reading the response, by default is `4096`.

  `headers`
  : Request headers object.

  `max_response_body_size`
  : The maximum size of the response body in bytes, by default is `32768`.

  `method`
  : HTTP method, by default the `GET` method is used.

  `verify`
  : Enables or disables verification of the HTTPS server certificate, by default is `true` (0.7.0).

  Example:

```
let reply = await ngx.fetch('http://example.com/');
let body = await reply.text();

r.return(200, body);
```

`ngx.log(level, message)`
: Writes a message to the error log with the specified level of logging. The `level` parameter specifies one of the log levels; the `message` parameter can be a string or Buffer. The following log levels can be specified: `ngx.INFO`, `ngx.WARN`, and `ngx.ERR`.

> **Note**
>
> As Angie has a hardcoded maximum line length limit, only the first 2048 bytes of the string can be logged.

`ngx.prefix`
: A string containing the file path to Angie prefix — a directory that keeps server files (0.8.0).

`ngx.version`
: A string containing Angie version, for example: `1.25.0` (0.8.0).

`ngx.version_number`
: A number containing Angie version, for example: `1025000` (0.8.0).

`ngx.worker_id`
: A number that corresponds to Angie internal worker ID, the value is between `0` and the value specified in the `worker_processes` directive (0.8.0).

ngx.shared

The `ngx.shared` global object is available since 0.8.0.

SharedDict

- `ngx.shared.SharedDict.add()`
- `ngx.shared.SharedDict.capacity`
- `ngx.shared.SharedDict.clear()`
- `ngx.shared.SharedDict.delete()`
- `ngx.shared.SharedDict.freeSpace()`
- `ngx.shared.SharedDict.get()`
- `ngx.shared.SharedDict.has()`
- `ngx.shared.SharedDict.incr()`
- `ngx.shared.SharedDict.items()`
- `ngx.shared.SharedDict.keys()`
- `ngx.shared.SharedDict.name`
- `ngx.shared.SharedDict.pop()`
- `ngx.shared.SharedDict.replace()`
- `ngx.shared.SharedDict.set()`
- `ngx.shared.SharedDict.size()`
- `ngx.shared.SharedDict.type`

The shared dictionary object is available since 0.8.0. The shared dictionary name, type, and size are set with the `js_shared_dict_zone` directive in HTTP or Stream.

A `SharedDict()` object has the following properties and methods:

`ngx.shared.SharedDict.add(key, value [,timeout])`
: Sets the `value` for the specified `key` in the dictionary only if the key does not exist yet. The `key` argument is a string representing the key of the item to add; the `value` argument is the value of the item to add.

The optional `timeout` argument is specified in milliseconds and overrides the `timeout` parameter of the `js_shared_dict_zone` directive in HTTP or Stream (since 0.8.5). It can be useful when some keys are expected to have unique timeouts.

Returns `true` if the value has been successfully added to the `SharedDict` dictionary; `false` if the key already exists in the dictionary. Throws `SharedMemoryError` if there is not enough free space in the `SharedDict` dictionary. Throws `TypeError` if the `value` is of a different type than expected by this dictionary.

`ngx.shared.SharedDict.capacity`
Returns the capacity of the `SharedDict` dictionary, corresponds to the `size` parameter of the `js_shared_dict_zone` directive in HTTP or Stream.

`ngx.shared.SharedDict.clear()`
Removes all items from the `SharedDict` dictionary.

`ngx.shared.SharedDict.delete(key)`
Removes the item associated with the specified key from the `SharedDict` dictionary; `true` if the item in the dictionary existed and was removed, `false` otherwise.

`ngx.shared.SharedDict.freeSpace()`
Returns the free page size in bytes. If the size is zero, the `SharedDict` dictionary will still accept new values if there is space in the occupied pages.

`ngx.shared.SharedDict.get(key)`
Retrieves the item by its `key`; returns the value associated with the `key` or `undefined` if there is none.

`ngx.shared.SharedDict.has(key)`
Searches for an item by its `key`; returns `true` if such item exists or `false` otherwise.

`ngx.shared.SharedDict.incr(key,delta[[,init], timeout])`
Increments the integer value associated with the `key` by `delta`. The `key` argument is a string; the `delta` argument is the number to increment or decrement the value by. If the key does not exist, the item will be initialized to the optional `init` argument, by default is `0`.

The optional `timeout` argument is specified in milliseconds and overrides the `timeout` parameter of the `js_shared_dict_zone` directive in HTTP or Stream (since 0.8.5). It can be useful when some keys are expected to have unique timeouts.

Returns the new value. Throws `SharedMemoryError` if there is not enough free space in the `SharedDict` dictionary. Throws `TypeError` if this dictionary does not expect numbers.

> **Note**
>
> This method can be used only if the dictionary type was declared with the `type=number` parameter of the `js_shared_dict_zone` directive in HTTP or Stream.

`ngx.shared.SharedDict.items([maxCount])`
Returns an array of the `SharedDict` dictionary key-value items (since 0.8.1). The `maxCount` parameter sets the maximum number of items to retrieve, by default is `1024`.

`ngx.shared.SharedDict.keys([maxCount])`
Returns an array of the `SharedDict` dictionary keys. The `maxCount` parameter sets the maximum number of keys to retrieve, by default is `1024`.

`ngx.shared.SharedDict.name`
Returns the name of the `SharedDict` dictionary, corresponds to the `zone=` parameter of the `js_shared_dict_zone` directive in HTTP or Stream.

`ngx.shared.SharedDict.pop(key)`
Removes the item associated with the specified `key` from the `SharedDict` dictionary; returns the value associated with the `key` or `undefined` if there is none.

`ngx.shared.SharedDict.replace(key, value)`
: Replaces the `value` for the specified `key` only if the key already exists; returns `true` if the value was successfully replaced, `false` if the key does not exist in the `SharedDict` dictionary. Throws `SharedMemoryError` if there is not enough free space in the `SharedDict` dictionary. Throws `TypeError` if the `value` is of a different type than expected by this dictionary.

`ngx.shared.SharedDict.set(key, value [,timeout])`
: Sets the `value` for the specified `key`; returns this `SharedDict` dictionary (for method chaining).

  The optional `timeout` argument is specified in milliseconds and overrides the `timeout` parameter of the `js_shared_dict_zone` directive in HTTP or Stream (since 0.8.5). It can be useful when some keys are expected to have unique timeouts.

`ngx.shared.SharedDict.size()`
: Returns the number of items for the `SharedDict` dictionary.

`ngx.shared.SharedDict.type`
: Returns `string` or `number` that corresponds to the `SharedDict` dictionary type set by the `type=` parameter of the `js_shared_dict_zone` directive in HTTP or Stream.

Built-in Objects

console

- `console.error()`
- `console.info()`
- `console.log()`
- `console.time()`
- `console.timeEnd()`
- `console.warn()`

The `console` object is available in Angie since 0.8.2, in CLI since 0.2.6.

`console.error(msg[, msg2 ...])`
: Outputs one or more error messages. The message may be a string or an object.

`console.info(msg[, msg2 ...])`
: Outputs one or more info messages. The message may be a string or an object.

`console.log(msg[, msg2 ...])`
: Outputs one or more log messages. The message may be a string or an object.

`console.time(label)`
: Starts a timer that can track how long an operation takes. The `label` parameter allows naming different timers. If `console.timeEnd()` is called with the same name, the time that elapsed since the timer was started will be output, in milliseconds.

`console.timeEnd(label)`
: Stops a timer previously started by `console.time()`. The `label` parameter allows naming different timers.

`console.warn(msg[, msg2 ...])`
: Outputs one or more warning messages. The message may be a string or an object.

crypto

- `crypto.getRandomValues()`
- `crypto.subtle.encrypt()`
- `crypto.subtle.decrypt()`
- `crypto.subtle.deriveBits()`

- `crypto.subtle.deriveKey()`
- `crypto.subtle.digest()`
- `crypto.subtle.exportKey()`
- `crypto.subtle.generateKey()`
- `crypto.subtle.importKey()`
- `crypto.subtle.sign()`
- `crypto.subtle.verify()`

The `crypto` object is a global object that allows using cryptographic functionality (since 0.7.0).

`crypto.getRandomValues(typedArray)`
: Gets cryptographically strong random values. Returns the same array passed as `typedArray` but with its contents replaced with the newly generated random numbers. Possible values:

  `typedArray`
  : Can be `Int8Array`, `Int16Array`, `Uint16Array`, `Int32Array`, or `Uint32Array`.

`crypto.subtle.encrypt(algorithm, key, data)`
: Encrypts `data` using the provided `algorithm` and `key`. Returns a `Promise` that fulfills with an `ArrayBuffer` containing the ciphertext. Possible values:

  `algorithm`
  : An object that specifies the algorithm to be used and any extra parameters if required:

    - For `RSA-OAEP`, pass an object with the following keys:

      `name`
      : A string, should be set to `RSA-OAEP`:

      ```
      crypto.subtle.encrypt({name: "RSA-OAEP"}, key, data)
      ```

    - For `AES-CTR`, pass an object with the following keys:

      `name`
      : A string, should be set to `AES-CTR`.

      `counter`
      : An `ArrayBuffer`, `TypedArray`, or `DataView` — the initial value of the counter block, must be 16 bytes long (the AES block size). The rightmost length bits of this block are used for the counter, and the rest is used for the nonce. For example, if length is set to 64, then the first half of counter is the nonce and the second half is used for the counter.

      `length`
      : The number of bits in the counter block that are used for the actual counter. The counter must be big enough that it doesn't wrap.

    - For `AES-CBC`, pass an object with the following keys:

      `name`
      : A string, should be set to `AES-CBC`.

      `iv`
      : The initialization vector, is an `ArrayBuffer`, `TypedArray`, or `DataView`, must be 16 bytes, unpredictable, and preferably cryptographically random. However, it need not be secret, for example, it may be transmitted unencrypted along with the ciphertext.

    - For `AES-GCM`, pass an object with the following keys:

      `name`
      : A string, should be set to `AES-GCM`.

`iv`
: The initialization vector, is an `ArrayBuffer`, `TypedArray`, or `DataView`, must be 16 bytes, and must be unique for every encryption operation carried out with a given key.

`additionalData`
: (optional) is an `ArrayBuffer`, `TypedArray`, or `DataView` that contains additional data that will not be encrypted but will be authenticated along with the encrypted data. If `additionalData` is specified, then the same data must be specified in the corresponding call to `decrypt()`: if the data given to the `decrypt()` call does not match the original data, the decryption will throw an exception. The bit length of `additionalData` must be smaller than `2^64 - 1`.

`tagLength`
: (optional, default is `128`) - a `number` that determines the size in bits of the authentication tag generated in the encryption operation and used for authentication in the corresponding decryption. Possible values: `32`, `64`, `96`, `104`, `112`, `120`, or `128`. The AES-GCM specification recommends that it should be `96`, `104`, `112`, `120`, or `128`, although `32` or `64` bits may be acceptable in some applications.

`key`
: A `CryptoKey` that contains the key to be used for encryption.

`data`
: An `ArrayBuffer`, `TypedArray`, or `DataView` that contains the data to be encrypted (also known as the plaintext).

`crypto.subtle.decrypt(algorithm, key, data)`
: Decrypts encrypted data. Returns a `Promise` with the decrypted data. Possible values:

`algorithm`
: An object that specifies the algorithm to be used, and any extra parameters as required. The values given for the extra parameters must match those passed into the corresponding `encrypt()` call.

- For `RSA-OAEP`, pass an object with the following keys:

  `name`
  : A string, should be set to `RSA-OAEP`:

  ```
  crypto.subtle.encrypt({name: "RSA-OAEP"}, key, data)
  ```

- For `AES-CTR`, pass an object with the following keys:

  `name`
  : A string, should be set to `AES-CTR`.

  `counter`
  : An `ArrayBuffer`, `TypedArray`, or `DataView` — the initial value of the counter block, must be 16 bytes long (the AES block size). The rightmost length bits of this block are used for the counter, and the rest is used for the nonce. For example, if length is set to 64, then the first half of counter is the nonce and the second half is used for the counter.

  `length`
  : The number of bits in the counter block that are used for the actual counter. The counter must be big enough that it doesn't wrap.

- For `AES-CBC`, pass an object with the following keys:

  `name`
  : A string, should be set to `AES-CBC`.

  `iv`
  : The initialization vector, is an `ArrayBuffer`, `TypedArray`, or `DataView`, must be 16

bytes, unpredictable, and preferably cryptographically random. However, it need not be secret (for example, it may be transmitted unencrypted along with the ciphertext).

- For `AES-GCM`, pass an object with the following keys:

  `name`
  : A string, should be set to `AES-GCM`.

  `iv`
  : The initialization vector, is an `ArrayBuffer`, `TypedArray`, or `DataView`, must be 16 bytes, and must be unique for every encryption operation carried out with a given key.

  `additionalData`
  : (optional) is an `ArrayBuffer`, `TypedArray`, or `DataView` that contains additional data that will not be encrypted but will be authenticated along with the encrypted data. If `additionalData` is specified, then the same data must be specified in the corresponding call to `decrypt()`: if the data given to the `decrypt()` call does not match the original data, the decryption will throw an exception. The bit length of `additionalData` must be smaller than `2^64 - 1`.

  `tagLength`
  : (optional, default is `128`) - a `number` that determines the size in bits of the authentication tag generated in the encryption operation and used for authentication in the corresponding decryption. Possible values: `32`, `64`, `96`, `104`, `112`, `120`, or `128`. The AES-GCM specification recommends that it should be `96`, `104`, `112`, `120`, or `128`, although `32` or `64` bits may be acceptable in some applications.

`key`
: A `CryptoKey` that contains the key to be used for decryption. If `RSA-OAEP` is used, this is the `privateKey` property of the `CryptoKeyPair` object.

`data`
: An `ArrayBuffer`, `TypedArray`, or `DataView` that contains the data to be decrypted (also known as ciphertext).

`crypto.subtle.deriveBits(algorithm, baseKey, length)`
: Derives an array of bits from a base key. Returns a `Promise` which will be fulfilled with an `ArrayBuffer` containing the derived bits. Possible values:

  `algorithm`
  : An object that defines the derivation algorithm to use:

    - For `HKDF`, pass an object with the following keys:

      `name`
      : A string, should be set to `HKDF`.

      `hash`
      : A string with the digest algorithm to use: `SHA-1`, `SHA-256`, `SHA-384`, or `SHA-512`.

      `salt`
      : An `ArrayBuffer`, `TypedArray`, or `DataView` that represents a random or pseudo-random value with the same length as the output of the `digest` function. Unlike the input key material passed into `deriveKey()`, salt does not need to be kept secret.

      `info`
      : An `ArrayBuffer`, `TypedArray`, or `DataView` that represents application-specific contextual information used to bind the derived key to an application or context, and enables deriving different keys for different contexts while using the same input key material. This property is required but may be an empty buffer.

    - For `PBKDF2`, pass an object with the following keys:

      `name`
      : A string, should be set to `PBKDF2`.

`hash`
A string with the digest algorithm to use: `SHA-1`, `SHA-256`, `SHA-384`, or `SHA-512`.

`salt`
An `ArrayBuffer`, `TypedArray`, or `DataView` that represents random or pseudo-random value of at least `16` bytes. Unlike the input key material passed into `deriveKey()`, salt does not need to be kept secret.

`iterations`
A `number` that represents the number of times the hash function will be executed in `deriveKey()`.

- For `ECDH`, pass the object with the following keys (since 0.9.1):

`name`
A string, should be set to `ECDH`.

`public`
A `CryptoKey` that represents the public key of the other party. The key must be generated using the same curve as the base key.

`baseKey`
A `CryptoKey` that represents the input to the derivation algorithm - the initial key material for the derivation function: for example, for `PBKDF2` it might be a password, imported as a `CryptoKey` using `crypto.subtle.importKey()`.

`length`
A number representing the number of bits to derive. For browser compatibility, the number should be a multiple of `8`.

`crypto.subtle.deriveKey(algorithm, baseKey, derivedKeyAlgorithm, extractable, keyUsages)`
Derives a secret key from a master key. Possible values:

`algorithm`
An object that defines the derivation algorithm to use:

- For `HKDF`, pass the object with the following keys:

`name`
A string, should be set to `HKDF`.

`hash`
A string with the digest algorithm to use: `SHA-1`, `SHA-256`, `SHA-384`, or `SHA-512`.

`salt`
An `ArrayBuffer`, `TypedArray`, or `DataView` that represents random or pseudo-random value with the same length as the output of the `digest` function. Unlike the input key material passed into `deriveKey()`, salt does not need to be kept secret.

`info`
An `ArrayBuffer`, `TypedArray`, or `DataView` that represents application-specific contextual information used to bind the derived key to an application or context, and enables deriving different keys for different contexts while using the same input key material. This property is required but may be an empty buffer.

- For `PBKDF2`, pass the object with the following keys:

`name`
A string, should be set to `PBKDF2`.

`hash`
A string with the digest algorithm to use: `SHA-1`, `SHA-256`, `SHA-384`, or `SHA-512`.

`salt`
An `ArrayBuffer`, `TypedArray`, or `DataView` that represents random or pseudo-

random value of at least `16` bytes. Unlike the input key material passed into `deriveKey()`, salt does not need to be kept secret.

`iterations`
A `number` that represents the number of times the hash function will be executed in `deriveKey()`.

- For `ECDH`, pass the object with the following keys (since 0.9.1):

`name`
A string, should be set to `ECDH`.

`publicKey`
A `CryptoKey` that represents the public key of the other party. The key must be generated using the same curve as the base key.

`baseKey`
A `CryptoKey` that represents the input to the derivation algorithm - the initial key material for the derivation function: for example, for `PBKDF2` it might be a password, imported as a `CryptoKey` using `crypto.subtle.importKey()`.

`derivedKeyAlgorithm`
An object that defines the algorithm the derived key will be used for:

- For `HMAC`, pass the object with the following keys:

`name`
A string, should be set to `HMAC`.

`hash`
A string with the name of the digest function to use: `SHA-1`, `SHA-256`, `SHA-384`, or `SHA-512`.

`length`
(optional) is a `number` that represents the length in bits of the key. If not specified, the length of the key is equal to the block size of the chosen hash function.

- For `AES-CTR`, `AES-CBC`, or `AES-GCM`, pass the object with the following keys:

`name`
A string, should be set to `AES-CTR`, `AES-CBC`, or `AES-GCM`, depending on the algorithm used.

`length`
A `number` that represents the length in bits of the key to generate: `128`, `192`, or `256`.

`extractable`
A boolean value that indicates whether it will be possible to export the key.

`keyUsages`
An `Array` that indicates what can be done with the derived key. The key usages must be allowed by the algorithm set in `derivedKeyAlgorithm`. Possible values:

`encrypt`
Key for encrypting messages.

`decrypt`
Key for decrypting messages.

`sign`
Key for signing messages.

`verify`
Key for verifying signatures.

`deriveKey`
Key for deriving a new key.

`deriveBits`
: Key for deriving bits.

`wrapKey`
: Key for wrapping a key.

`unwrapKey`
: Key for unwrapping a key.

`crypto.subtle.digest(algorithm, data)`
: Generates a digest of the given data. Takes as its arguments an identifier for the digest algorithm to use and the data to digest. Returns a `Promise` which will be fulfilled with the digest. Possible values:

  `algorithm`
  : A string that defines the hash function to use: `SHA-1` (not for cryptographic applications), `SHA-256`, `SHA-384`, or `SHA-512`.

  `data`
  : An `ArrayBuffer`, `TypedArray`, or `DataView` that contains the data to be digested.

`crypto.subtle.exportKey(format, key)`
: Exports a key: takes a key as a `CryptoKey` object and returns the key in an external, portable format (since 0.7.10). If the `format` was `jwk`, then the `Promise` fulfills with a JSON object containing the key. Otherwise, the promise fulfills with an `ArrayBuffer` containing the key. Possible values:

  `format`
  : A string that describes the data format in which the key should be exported, can be the following:

    `raw`
    : The raw data format.

    `pkcs8`
    : The PKCS #8 format.

    `spki`
    : The SubjectPublicKeyInfo format.

    `jwk`
    : The JSON Web Key (JWK) format (since 0.7.10).

  `key`
  : The `CryptoKey` that contains the key to be exported.

`crypto.subtle.generateKey(algorithm, extractable, usage)`
: Generates a new key for symmetric algorithms or key pair for public-key algorithms (since 0.7.10). Returns a `Promise` that fulfills with the generated key as a `CryptoKey` or `CryptoKeyPair` object. Possible values:

  `algorithm`
  : A dictionary object that defines the type of key to generate and provides extra algorithm-specific parameters:

    - For `RSASSA-PKCS1-v1_5`, `RSA-PSS`, or `RSA-OAEP`, pass the object with the following keys:

      `name`
      : A string, should be set to `RSASSA-PKCS1-v1_5`, `RSA-PSS`, or `RSA-OAEP`, depending on the algorithm used.

      `hash`
      : A string that represents the name of the `digest` function to use, can be `SHA-256`, `SHA-384`, or `SHA-512`.

    - For `ECDSA`, pass the object with the following keys:

      `name`
      : A string, should be set to `ECDSA`.

`namedCurve`
A string that represents the name of the elliptic curve to use, may be `P-256`, `P-384`, or `P-521`.

- For `HMAC`, pass the object with the following keys:

  `name`
  A string, should be set to `HMAC`.

  `hash`
  A string that represents the name of the `digest` function to use, can be `SHA-256`, `SHA-384`, or `SHA-512`.

  `length`
  (optional) is a number that represents the length in bits of the key. If omitted, the length of the key is equal to the length of the digest generated by the chosen digest function.

- For `AES-CTR`, `AES-CBC`, or `AES-GCM`, pass the string identifying the algorithm or an object of the form *`"name": "ALGORITHM"`* , where `ALGORITHM` is the name of the algorithm.

- For `ECDH`, pass the object with the following keys (since 0.9.1):

  `name`
  A string, should be set to `ECDH`.

  `namedCurve`
  A string that represents the name of the elliptic curve to use, may be `P-256`, `P-384`, or `P-521`.

`extractable`
Boolean value that indicates if it is possible to export the key.

`usage`
An `array` that indicates possible actions with the key:

`encrypt`
Key for encrypting messages.

`decrypt`
Key for decrypting messages.

`sign`
Key for signing messages.

`verify`
Key for verifying signatures.

`deriveKey`
Key for deriving a new key.

`deriveBits`
Key for deriving bits.

`wrapKey`
Key for wrapping a key.

`unwrapKey`
Key for unwrapping a key.

`crypto.subtle.importKey(format, keyData, algorithm, extractable, keyUsages)`
Imports a key: takes as input a key in an external, portable format and gives a `CryptoKey` object. Returns a `Promise` that fulfills with the imported key as a `CryptoKey` object. Possible values:

`format`
A string that describes the data format of the key to import, can be the following:

`raw`
The raw data format.

`pkcs8`
: The PKCS #8 format.

`spki`
: The SubjectPublicKeyInfo format.

`jwk`
: The JSON Web Key (JWK) format (since 0.7.10).

`keyData`
: The `ArrayBuffer`, `TypedArray`, or `DataView` object that contains the key in the given format.

`algorithm`
: A dictionary object that defines the type of key to import and provides extra algorithm-specific parameters:

- For `RSASSA-PKCS1-v1_5`, `RSA-PSS`, or `RSA-OAEP`, pass the object with the following keys:

  `name`
  : A string, should be set to `RSASSA-PKCS1-v1_5`, `RSA-PSS`, or `RSA-OAEP`, depending on the used algorithm.

  `hash`
  : A string that represents the name of the `digest` function to use, can be `SHA-1`, `SHA-256`, `SHA-384`, or `SHA-512`.

- For `ECDSA`, pass the object with the following keys:

  `name`
  : A string, should be set to `ECDSA`.

  `namedCurve`
  : A string that represents the name of the elliptic curve to use, may be `P-256`, `P-384`, or `P-521`.

- For `HMAC`, pass the object with the following keys:

  `name`
  : A string, should be set to `HMAC`.

  `hash`
  : A string that represents the name of the `digest` function to use, can be `SHA-256`, `SHA-384`, or `SHA-512`.

  `length`
  : (optional) is a number that represents the length in bits of the key. If omitted, the length of the key is equal to the length of the digest generated by the chosen digest function.

- For `AES-CTR`, `AES-CBC`, or `AES-GCM`, pass the string identifying the algorithm or an object of the form *`"name": "ALGORITHM"`* , where `ALGORITHM` is the name of the algorithm.
- For `PBKDF2`, pass the `PBKDF2` string.
- For `HKDF`, pass the `HKDF` string.
- For `ECDH`, pass the object with the following keys (since 0.9.1):

  `name`
  : A string, should be set to `ECDH`.

  `namedCurve`
  : A string that represents the name of the elliptic curve to use, may be `P-256`, `P-384`, or `P-521`.

`extractable`
: Boolean value that indicates if it is possible to export the key.

`keyUsages`
An `array` that indicates possible actions with the key:

`encrypt`
Key for encrypting messages.

`decrypt`
Key for decrypting messages.

`sign`
Key for signing messages.

`verify`
Key for verifying signatures.

`deriveKey`
Key for deriving a new key.

`deriveBits`
Key for deriving bits.

`wrapKey`
Key for wrapping a key.

`unwrapKey`
Key for unwrapping a key.

`crypto.subtle.sign(algorithm, key, data)`
Returns `signature` as a `Promise` that fulfills with an `ArrayBuffer` containing the signature. Possible values:

`algorithm`
A string or object that specifies the signature algorithm to use and its parameters:

- For `RSASSA-PKCS1-v1_5`, pass the string identifying the algorithm or an object of the form *`"name": "ALGORITHM"`* .
- For `RSA-PSS`, pass the object with the following keys:

  `name`
  A string, should be set to `RSA-PSS`.

  `saltLength`
  A long `integer` that represents the length of the random salt to use, in bytes.
- For `ECDSA`, pass the object with the following keys:

  `name`
  A string, should be set to `ECDSA`.

  `hash`
  An identifier for the digest algorithm to use, can be `SHA-256`, `SHA-384`, or `SHA-512`.
- For `HMAC`, pass the string identifying the algorithm or an object of the form *`"name": "ALGORITHM"`* .

`key`
A `CryptoKey` object that contains the key to be used for signing. If algorithm identifies a public-key cryptosystem, this is the private key.

`data`
An `ArrayBuffer`, `TypedArray`, or `DataView` object that contains the data to be signed.

`crypto.subtle.verify(algorithm, key, signature, data)`
Verifies a digital signature; returns a `Promise` that fulfills with a boolean value: `true` if the signature is valid, otherwise `false`. Possible values:

`algorithm`
A string or object that specifies the algorithm to use and its parameters:

- For `RSASSA-PKCS1-v1_5`, pass the string identifying the algorithm or an object of the form *`"name": "ALGORITHM"`*.
- For `RSA-PSS`, pass the object with the following keys:

  `name`
  : A string, should be set to `RSA-PSS`.

  `saltLength`
  : A long `integer` that represents the length of the random salt to use, in bytes.

- For `ECDSA`, pass the object with the following keys:

  `name`
  : A string, should be set to `ECDSA`.

  `hash`
  : An identifier for the digest algorithm to use, can be `SHA-256`, `SHA-384`, or `SHA-512`.

- For `HMAC`, pass the string identifying the algorithm or an object of the form *`"name": "ALGORITHM"`*.

`key`
: A `CryptoKey` object that contains the key to be used for verifying. It is the secret key for a symmetric algorithm and the public key for a public-key system.

`signature`
: An `ArrayBuffer`, `TypedArray`, or `DataView` that contains the signature to verify.

`data`
: An `ArrayBuffer`, `TypedArray`, or `DataView` object that contains the data whose signature is to be verified.

CryptoKey

- `CryptoKey.algorithm`
- `CryptoKey.extractable`
- `CryptoKey.type`
- `CryptoKey.usages`

The `CryptoKey` object represents a cryptographic `key` obtained from one of the `SubtleCrypto` methods: `crypto.subtle.generateKey()`, `crypto.subtle.deriveKey()`, `crypto.subtle.importKey()`.

`CryptoKey.algorithm`
: Returns an object describing the algorithm for which this key can be used and any associated extra parameters (since 0.8.0), read-only.

`CryptoKey.extractable`
: A boolean value, `true` if the key can be exported (since 0.8.0), read-only.

`CryptoKey.type`
: A string value that indicates which kind of key is represented by the object, read-only. Possible values:

  `secret`
  : This key is a secret key for use with a symmetric algorithm.

  `private`
  : This key is the private half of an asymmetric algorithm's `CryptoKeyPair`.

  `public`
  : This key is the public half of an asymmetric algorithm's `CryptoKeyPair`.

`CryptoKey.usages`
: An array of strings indicating what this key can be used for (since 0.8.0), read-only. Possible array values:

`encrypt`
: Key for encrypting messages.

`decrypt`
: Key for decrypting messages.

`sign`
: Key for signing messages.

`verify`
: Key for verifying signatures.

`deriveKey`
: Key for deriving a new key.

`deriveBits`
: Key for deriving bits.

CryptoKeyPair

- `CryptoKeyPair.privateKey`
- `CryptoKeyPair.publicKey`

The `CryptoKeyPair` is a dictionary object of the WebCrypto API that represents an asymmetric key pair.

`CryptoKeyPair.privateKey`
: A `CryptoKey` object representing the private key.

`CryptoKeyPair.publicKey`
: A `CryptoKey` object representing the public key.

njs

- `njs.version`
- `njs.version_number`
- `njs.dump()`
- `njs.memoryStats`
- `njs.on()`

The `njs` object is a global object that represents the current VM instance (since 0.2.0).

`njs.version`
: Returns a string with the current version of NJS (for example, "0.7.4").

`njs.version_number`
: Returns a number with the current version of NJS. For example, "0.7.4" is returned as `0x000704` (since 0.7.4).

`njs.dump(value)`
: Returns the pretty-print string representation for a value.

`njs.memoryStats`
: Object containing memory statistics for the current VM instance (since 0.7.8).

  `size`
  : Amount of memory in bytes NJS memory pool claimed from the operating system.

`njs.on(event, callback)`
: Registers a callback for the specified VM event (since 0.5.2). An event may be one of the following strings:

  `exit`
  : Is called before the VM is destroyed. The callback is called without arguments.

process

- `process.argv`
- `process.env`
- `process.kill()`
- `process.pid`
- `process.ppid`

The `process` object is a global object that provides information about the current process (0.3.3).

`process.argv`
: Returns an array that contains the command-line arguments passed when the current process was launched.

`process.env`
: Returns an object containing the user environment.

> **Note**
>
> By default, Angie removes all environment variables inherited from its parent process except the TZ variable. Use the `env` directive to preserve some of the inherited variables.

`process.kill(pid, number | string)`
: Sends the signal to the process identified by `pid`. Signal names are numbers or strings such as `SIGINT` or `SIGHUP`. See kill(2) for more information.

`process.pid`
: Returns the PID of the current process.

`process.ppid`
: Returns the PID of the current parent process.

String

By default, all strings in NJS are Unicode strings. They correspond to ECMAScript strings that contain Unicode characters. Before 0.8.0, byte strings were also supported.

Byte Strings (Removed)

> **Note**
>
> Since 0.8.0, the support for byte strings and byte string methods was removed. When working with byte sequences, the *Buffer* object and Buffer properties, such as `r.requestBuffer`, `r.rawVariables`, should be used.

Byte strings contained a sequence of bytes and were used to serialize Unicode strings to external data and deserialize from external sources. For example, the `toUTF8()` method serialized a Unicode string to a byte string using UTF-8 encoding. The `toBytes()` method serialized a Unicode string with code points up to 255 into a byte string; otherwise, `null` was returned.

The following methods were made obsolete and removed in 0.8.0:

- `String.bytesFrom()` (removed in 0.8.0, use `Buffer.from()`)
- `String.prototype.fromBytes()` (removed in 0.8.0)
- `String.prototype.fromUTF8()` (removed in 0.8.0, use `TextDecoder`)
- `String.prototype.toBytes()` (removed in 0.8.0)

- `String.prototype.toString()` with encoding (removed in 0.8.0)
- `String.prototype.toUTF8()` (removed in 0.8.0, use `TextEncoder`)

Web API

TextDecoder

- `TextDecoder()`
- `TextDecoder.prototype.encoding`
- `TextDecoder.prototype.fatal`
- `TextDecoder.prototype.ignoreBOM`
- `TextDecoder.prototype.decode()`

The `TextDecoder` produces a stream of code points from a stream of bytes (0.4.3).

`TextDecoder([[encoding], options])`
: Creates a new `TextDecoder` object for the specified `encoding`; currently, only UTF-8 is supported. The `options` is a `TextDecoderOptions` dictionary with the property:

  `fatal`
  : Boolean flag indicating if `TextDecoder.decode()` must throw the `TypeError` exception when a coding error is found, by default is `false`.

`TextDecoder.prototype.encoding`
: Returns a string with the name of the encoding used by `TextDecoder()`, read-only.

`TextDecoder.prototype.fatal`
: Boolean flag, `true` if the error mode is fatal, read-only.

`TextDecoder.prototype.ignoreBOM`
: Boolean flag, `true` if the byte order marker is ignored, read-only.

`TextDecoder.prototype.decode(buffer, [options])`
: Returns a string with the text decoded from the `buffer` by `TextDecoder()`. The buffer can be `ArrayBuffer`. The `options` is a `TextDecodeOptions` dictionary with the property:

  `stream`
  : Boolean flag indicating if additional data will follow in subsequent calls to `decode()`: `true` if processing the data in chunks, and `false` for the final chunk or if the data is not chunked. By default is `false`.

  Example:

```
>> (new TextDecoder()).decode(new Uint8Array([206,177,206,178]))
αβ
```

TextEncoder

- `TextEncoder()`
- `TextEncoder.prototype.encode()`
- `TextEncoder.prototype.encodeInto()`

The `TextEncoder` object produces a byte stream with UTF-8 encoding from a stream of code points (0.4.3).

`TextEncoder()`
: Returns a newly constructed `TextEncoder` that will generate a byte stream with UTF-8 encoding.

`TextEncoder.prototype.encode(string)`
: Encodes `string` into a `Uint8Array` with UTF-8 encoded text.

`TextEncoder.prototype.encodeInto(string, uint8Array)`
: Encodes a `string` to UTF-8, puts the result into the destination `Uint8Array`, and returns a dictionary object that shows the progress of the encoding. The dictionary object contains two members:

  `read`
  : The number of UTF-16 units of code from the source `string` converted to UTF-8.

  `written`
  : The number of bytes modified in the destination `Uint8Array`.

Timers

- `clearTimeout()`
- `setTimeout()`

`clearTimeout(timeout)`
: Cancels a `timeout` object created by `setTimeout()`.

`setTimeout(function, milliseconds[, argument1, argumentN])`
: Calls a `function` after a specified number of `milliseconds`. One or more optional `arguments` can be passed to the specified function. Returns a `timeout` object.

  Example:

```
function handler(v)
{
    // ...
}

t = setTimeout(handler, 12);

// ...

clearTimeout(t);
```

Global Functions

- `atob()`
- `btoa()`

`atob(encodedData)`
: Decodes a string of data which has been encoded using `Base64` encoding. The `encodedData` parameter is a binary string that contains Base64-encoded data. Returns a string that contains decoded data from `encodedData`.

  The similar `btoa()` method can be used to encode and transmit data which may otherwise cause communication problems, then transmit it and use the `atob()` method to decode the data again. For example, you can encode, transmit, and decode control characters such as ASCII values `0` through `31`.

  Example:

```
const encodedData = btoa("text to encode"); // encode a string
const decodedData = atob(encodedData); // decode the string
```

`btoa(stringToEncode)`
: Creates a Base64-encoded ASCII string from a binary string. The `stringToEncode` parameter is a binary string to encode. Returns an ASCII string containing the Base64 representation of `stringToEncode`.

The method can be used to encode data which may otherwise cause communication problems, transmit it, then use the `atob()` method to decode the data again. For example, you can encode control characters such as ASCII values `0` through `31`.

Example:

```
const encodedData = btoa("text to encode"); // encode a string
const decodedData = atob(encodedData); // decode the string
```

Built-in Modules

Buffer

The `Buffer` object is a Node.js-compatible way to work with binary data. Due to the file's extensive size, this section is limited to a comprehensive list of Buffer methods.

- `Buffer.alloc()`
- `Buffer.allocUnsafe()`
- `Buffer.byteLength()`
- `Buffer.compare()`
- `Buffer.concat()`
- `Buffer.from(array)`
- `Buffer.from(arrayBuffer)`
- `Buffer.from(buffer)`
- `Buffer.from(object)`
- `Buffer.from(string)`
- `Buffer.isBuffer()`
- `Buffer.isEncoding()`
- `buffer[]`
- `buf.buffer`
- `buf.byteOffset`
- `buf.compare()`
- `buf.copy()`
- `buf.equals()`
- `buf.fill()`
- `buf.includes()`
- `buf.indexOf()`
- `buf.lastIndexOf()`
- `buf.length`
- `buf.readIntBE()`
- `buf.readIntLE()`
- `buf.readUIntBE()`
- `buf.readUIntLE()`
- `buf.readDoubleBE()`
- `buf.readDoubleLE()`

- `buf.readFloatBE()`
- `buf.readFloatLE()`
- `buf.subarray()`
- `buf.slice()`
- `buf.swap16()`
- `buf.swap32()`
- `buf.swap64()`
- `buf.toJSON()`
- `buf.toString()`
- `buf.write()`
- `buf.writeIntBE()`
- `buf.writeIntLE()`
- `buf.writeUIntBE()`
- `buf.writeUIntLE()`
- `buf.writeDoubleBE()`
- `buf.writeDoubleLE()`
- `buf.writeFloatBE()`
- `buf.writeFloatLE()`

For detailed documentation of Buffer methods, please refer to Node.js Buffer documentation.

Crypto

The Crypto module provides cryptographic functionality support. The Crypto module object is imported using `import crypto from 'crypto'`.

Note

Since 0.7.0, extended crypto API is available as a global *crypto* object.

- `crypto.createHash()`
- `crypto.createHmac()`

`crypto.createHash(algorithm)`
: Creates and returns a Hash object that can be used to generate hash digests using the given `algorithm`. The algorithm can be `md5`, `sha1`, and `sha256`.

`crypto.createHmac(algorithm, secret key)`
: Creates and returns an HMAC object that uses the given `algorithm` and `secret key`. The algorithm can be `md5`, `sha1`, and `sha256`.

Hash

- `hash.update()`
- `hash.digest()`

`hash.update(data)`
: Updates the hash content with the given `data`.

`hash.digest([encoding])`
: Calculates the digest of all of the data passed using `hash.update()`. The encoding can be `hex`, `base64`, and `base64url`. If encoding is not provided, a Buffer object is returned (0.4.4).

> **Note**
>
> Before version 0.4.4, a byte string was returned instead of a Buffer object.

`hash.copy()`
: Makes a copy of the current state of the hash (since 0.7.12).

Example:

```
import crypto from 'crypto';

crypto.createHash('sha1').update('A').update('B').digest('base64url');
/* BtlFlCqiamG-GMPiK_GbvKjdK10 */
```

HMAC

- `hmac.update()`
- `hmac.digest()`

`hmac.update(data)`
: Updates the HMAC content with the given `data`.

`hmac.digest([encoding])`
: Calculates the HMAC digest of all of the data passed using `hmac.update()`. The encoding can be `hex`, `base64`, and `base64url`. If encoding is not provided, a Buffer object is returned (0.4.4).

> **Note**
>
> Before version 0.4.4, a byte string was returned instead of a Buffer object.

fs

The `fs` module provides operations with the file system. The module object is imported using `import fs from 'fs'`.

- `fs.accessSync()`
- `fs.appendFileSync()`
- `fs.mkdirSync()`
- `fs.readdirSync()`
- `fs.readFileSync()`
- `fs.realpathSync()`
- `fs.renameSync()`
- `fs.rmdirSync()`
- `fs.symlinkSync()`
- `fs.unlinkSync()`
- `fs.writeFileSync()`
- `fs.promises.readFile()`

- `fs.promises.appendFile()`
- `fs.promises.writeFile()`
- `fs.promises.readdir()`
- `fs.promises.mkdir()`
- `fs.promises.rmdir()`
- `fs.promises.rename()`
- `fs.promises.unlink()`
- `fs.promises.symlink()`
- `fs.promises.access()`
- `fs.promises.realpath()`

For detailed documentation of fs methods, please refer to Node.js fs documentation.

Query String

The Query String module provides methods for parsing and formatting URL query strings. The module object is imported using `import qs from 'querystring'`.

- `querystring.decode()`
- `querystring.encode()`
- `querystring.escape()`
- `querystring.parse()`
- `querystring.stringify()`
- `querystring.unescape()`

`querystring.decode()`
: An alias to `querystring.parse()`.

`querystring.encode()`
: An alias to `querystring.stringify()`.

`querystring.escape(string)`
: Performs URL percent-encoding of the `string` in a manner optimized for the requirements of URL query strings. The method is used by `querystring.stringify()` and should not be used directly.

`querystring.parse(string[, separator[, equal[, options]]])`
: Parses the `string` as a URL query string and returns an object. The optional `separator` parameter (default: `&`) specifies the substring for delimiting key-value pairs. The optional `equal` parameter (default: `=`) specifies the substring for delimiting keys and values. The optional `options` parameter is an object that may contain the following property:

  `decodeURIComponent`
  : Function to use when decoding percent-encoded characters in the query string, default: `querystring.unescape()`.

  `maxKeys`
  : The maximum number of keys to parse, default: `1000`. The `0` value removes limitations for counting keys.

  Example:

```
>> qs.parse('foo=bar&abc=xyz&abc=123')
{
    foo: 'bar',
    abc: ['xyz', '123']
}
```

`querystring.stringify(object[, separator[, equal[, options]]])`
: Produces a URL query string from the `object` by iterating through its own properties. The optional `separator` parameter (default: `&`) specifies the substring for delimiting key-value pairs. The optional `equal` parameter (default: `=`) specifies the substring for delimiting keys and values. The optional `options` parameter is an object that may contain the following property:

  `encodeURIComponent`
  : Function to use when converting URL-unsafe characters to percent-encoding in the query string, default: `querystring.escape()`.

  Example:

```
>> qs.stringify({ foo: 'bar', baz: ['qux', 'quux'], corge: '' })
'foo=bar&baz=qux&baz=quux&corge='
```

`querystring.unescape(string)`
: Performs decoding of URL percent-encoded characters in the `string`. The method is used by `querystring.parse()` and should not be used directly.

XML

- `xml.parse()`
- `xml.c14n()`
- `xml.exclusiveC14n()`
- `xml.serialize()`
- `xml.serializeToString()`
- `XMLDoc`
- `XMLNode`
- `XMLAttr`

The XML module allows working with XML documents (since 0.7.10). The XML module object is imported using `import xml from 'xml'`.

Example:

```
import xml from 'xml';

let data = `<note><to b="bar" a= "foo" >Tove</to><from>Jani</from></note>`;
let doc = xml.parse(data);

console.log(doc.note.to.$text) /* 'Tove' */
console.log(doc.note.to.$attr$b) /* 'bar' */
console.log(doc.note.$tags[1].$text) /* 'Jani' */

let dec = new TextDecoder();
let c14n = dec.decode(xml.exclusiveC14n(doc.note));
console.log(c14n) /* '<note><to a="foo" b="bar">Tove</to><from>Jani</from></note>' */

c14n = dec.decode(xml.exclusiveC14n(doc.note.to));
console.log(c14n) /* '<to a="foo" b="bar">Tove</to>' */

c14n = dec.decode(xml.exclusiveC14n(doc.note, doc.note.to /* excluding 'to' */));
console.log(c14n) /* '<note><from>Jani</from></note>' */
```

`parse(string | Buffer)`
: Parses a string or Buffer for an XML document; returns an `XMLDoc` wrapper object representing the parsed XML document.

`c14n(root_node[, excluding_node])`
: Canonicalizes `root_node` and its children according to Canonical XML Version 1.1. The `root_node` can be an `XMLNode` or `XMLDoc` wrapper object around XML structure. Returns a Buffer object that contains canonicalized output.

  `excluding_node`
  : Allows omitting from the output a part of the document.

`exclusiveC14n(root_node[, excluding_node[, withComments[,prefix_list]]])`
: Canonicalizes `root_node` and its children according to Exclusive XML Canonicalization Version 1.0.

  `root_node`
  : An `XMLNode` or `XMLDoc` wrapper object around XML structure.

  `excluding_node`
  : Allows omitting from the output a part of the document corresponding to the node and its children.

  `withComments`
  : A boolean value, `false` by default. If `true`, canonicalization corresponds to Exclusive XML Canonicalization Version 1.0. Returns a Buffer object that contains canonicalized output.

  `prefix_list`
  : An optional string with space-separated namespace prefixes for namespaces that should also be included in the output.

`serialize()`
: The same as `xml.c14n()` (since 0.7.11).

`serializeToString()`
: The same as `xml.c14n()` except it returns the result as a `string` (since 0.7.11).

`XMLDoc`
: An XMLDoc wrapper object around XML structure, the root node of the document.

  `doc.$root`
  : The document's root by its name or undefined.

  `doc.abc`
  : The first root tag named `abc` as an `XMLNode` wrapper object.

`XMLNode`
: An XMLNode wrapper object around an XML tag node.

  `node.abc`
  : The same as `node.$tag$abc`.

  `node.$attr$abc`
  : The node's attribute value of `abc`, writable since 0.7.11.

  `node.$attr$abc=xyz`
  : The same as `node.setAttribute('abc', xyz)` (since 0.7.11).

  `node.$attrs`
  : An `XMLAttr` wrapper object for all attributes of the node.

  `node.$name`
  : The name of the node.

  `node.$ns`
  : The namespace of the node.

`node.$parent`
: The parent node of the current node.

`node.$tag$abc`
: The first child tag of the node named `abc`, writable since 0.7.11.

`node.$tags`
: An array of all child tags.

`node.$tags = [node1, node2, ...]`
: The same as `node.removeChildren(); node.addChild(node1); node.addChild(node2)` (since 0.7.11).

`node.$tags$abc`
: All child tags named `abc` of the node, writable since 0.7.11.

`node.$text`
: The content of the node, writable since 0.7.11.

`node.$text = 'abc'`
: The same as `node.setText('abc')` (since 0.7.11).

`node.addChild(nd)`
: Adds an XMLNode as a child to the node (since 0.7.11). `nd` is recursively copied before adding to the node.

`node.removeAllAttributes()`
: Removes all attributes of the node (since 0.7.11).

`node.removeAttribute(attr_name)`
: Removes the attribute named `attr_name` (since 0.7.11).

`node.removeChildren(tag_name)`
: Removes all child tags named `tag_name` (since 0.7.11). If `tag_name` is absent, all child tags are removed.

`node.removeText()`
: Removes the node's text value (0.7.11).

`node.setAttribute(attr_name, value)`
: Sets a value for `attr_name` (since 0.7.11). When the value is `null`, the attribute named `attr_name` is deleted.

`node.setText(value)`
: Sets a text value for the node (since 0.7.11). When the value is `null`, the text of the node is deleted.

`XMLAttr`
: An XMLAttrs wrapper object around XML node attributes.

`attr.abc`
: The attribute value of `abc`.

zlib

The `zlib` module (0.5.2) provides compression and decompression functionality using zlib. The module object is imported using `import zlib from 'zlib'`.

- `zlib.constants`
- `zlib.deflateRawSync()`
- `zlib.deflateSync()`
- `zlib.inflateRawSync()`
- `zlib.inflateSync()`

`zlib.constants`
: Returns a dictionary of zlib constants.

`zlib.deflateRawSync(data[, options])`
: Compresses `data` using the Deflate algorithm without the zlib header.

`zlib.deflateSync(data[, options])`
: Compresses `data` using the Deflate algorithm.

`zlib.inflateRawSync(data[, options])`
: Decompresses `data` using the Deflate algorithm without the zlib header.

`zlib.inflateSync(data[, options])`
: Decompresses `data` using the Deflate algorithm.

The `options` parameter is an object that may contain the following properties:

`level`
: Compression level (default: `zlib.constants.Z_DEFAULT_COMPRESSION`).

`memLevel`
: Specifies how much memory should be allocated for the compression state (default: `zlib.constants.Z_DEFAULT_MEMLEVEL`).

`strategy`
: Tunes the compression algorithm (default: `zlib.constants.Z_DEFAULT_STRATEGY`).

`windowBits`
: Sets the window size (default: `zlib.constants.Z_DEFAULT_WINDOWBITS`).

`dictionary`
: Buffer containing the predefined compression dictionary.

`info`
: A boolean value, if `true`, returns an object with buffer and engine.

`chunkSize`
: Chunk size for compression (default: `zlib.constants.Z_DEFAULT_CHUNK`).

Example:

```
import zlib from 'zlib';

const deflated = zlib.deflateSync('Hello World!');
const inflated = zlib.inflateSync(deflated);

console.log(inflated.toString()); // 'Hello World!'
```

You can also use the short link service at https://angie.ws/ to quickly find individual directives and variables:

3.2.4 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`).

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 available at: https://angie.ws/en/hu/server.

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 Module 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 available at: https://angie.ws/en/su/server.

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 use 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*: https://angie.ws/en/s/\protect\T2A\textdollarupstream_session_time

For placeholder variables such as `$http_<HEADER>` or `$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 other topics:

- Certificate Chaining
- Combined Locations
- Compact Server
- Configuration
- Configuration File Reloading
- Configuration Hashes
- 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 Locations
- 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 Proxying

3.3 Documentation for AI assistants

The site ships machine-readable copies of every documentation page so that LLM-powered tools — Claude Code, Cursor, ChatGPT, and other agentic assistants — can ingest the content directly instead of scraping rendered HTML.

3.3.1 llms.txt and llms-full.txt

Each language subdomain serves an llms.txt sitemap with a short project description and a list of every page (title, absolute URL, annotation). Its companion `llms-full.txt` concatenates the full Markdown body of every page into a single file suitable for one-shot ingestion:

- https://en.angie.software/llms.txt
- https://en.angie.software/llms-full.txt

These URLs are also advertised in robots.txt via the `Llms:` directive, so LLM crawlers discover them automatically.

3.3.2 Markdown versions of pages

Every HTML page has a Markdown twin. To fetch the Markdown version of any documentation URL, replace the trailing slash with `.md`:

- HTML: https://en.angie.software/angie/docs/configuration/
- Markdown: https://en.angie.software/angie/docs/configuration.md

Markdown is generated from the same reStructuredText source as the HTML build, so the content always stays in sync.

3.3.3 Context7

Angie documentation is indexed on Context7, a registry that serves up-to-date library documentation to AI code editors through its MCP server. The English Angie listing is at https://context7.com/websites/en_angie_software_angie.

3.4 Instructions

Step-by-step instructions for specific aspects of configuring Angie are provided here.

3.4.1 ACME Configuration

The *ACME* module in Angie enables automatic certificate retrieval using the ACME protocol. The ACME protocol supports various domain verification methods (also called "validation"); this module implements *HTTP validation*, *DNS validation*, *ALPN 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* directives 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	User Requirements	Multi-domain	Wildcard Domains
HTTP Validation	Open port 80 (or the one specified in *acme_http_port*) for incoming connections on the Angie server.	✓	
DNS Validation	Open port 53 (or the one specified in *acme_dns_port*) for incoming connections on the Angie server. Set an NS record for the `_acme-challenge.` subdomain pointing to your Angie server.	✓	✓
ALPN Validation	Open port 443 (or the TLS port used by your server) for incoming connections on the Angie server.	✓	
Hook validation	Create an external service (script or application) that can, on request from Angie, update DNS records or serve a special response via the web server.	✓	✓

- **Configure SSL using the obtained certificate and key**: The module makes certificates and keys 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
```

Note

These files persist on disk between restarts. On startup, the client reuses a stored certificate that is still valid instead of requesting a new one, avoiding both the issuance delay and unnecessary requests to the CA, which may be subject to rate limits. In a container, place the storage directory (by default `/var/lib/angie/acme/`) on a persistent volume so that issued certificates survive container recreation; see running Angie in a container.

The ACME client requires an account on the CA server. To create and manage this account, the client uses a private key (`account.key`). If no key exists, it is generated at startup. The client then uses this key to register the account with the server.

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 file using the `account_key` parameter in *acme_client*.

The ACME client also uses a separate key (`private.key`) for Certificate Signing Requests (CSRs). This certificate key is automatically created at startup if needed.

At startup, the client requests a certificate if one doesn't exist, signing and sending a CSR for all domains under its management to the CA server. The server verifies domain ownership using *HTTP* or *DNS validation* and issues a certificate, which the client saves locally (`certificate.pem`).

As mentioned earlier, a single certificate covers all domain names managed by the same ACME client, potentially resulting in a multi-domain certificate. The list of all names covered by the certificate can be found in the *Subject Alternative Name* (SAN) section of the obtained certificate. To check this from the command line:

```
$ openssl x509 -in certificate.pem -noout -text | grep -A5 "Subject Alternative Name"
```

When a certificate is about to expire or the domain list changes, the client signs and sends another CSR to the CA server. The server re-verifies ownership and issues a new certificate, which the client installs locally, replacing the previous one.

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 *ssl_certificate* and *ssl_certificate_key* directives:

```
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;
}
```

Note

The ACME client resolves the CA server's host name through the configured *resolver*, which must be present in the same context. On a host without IPv6 connectivity, the resolver may still issue AAAA (IPv6) queries for the CA and fail to connect; disable them with the `ipv6=off` parameter, for example `resolver 127.0.0.53 ipv6=off;`.

Domain Collection vs. Certificate Usage

The *acme* directive serves only to collect domain names for certificate requests. It does not control where the certificate can be used: any `server` block can reference the obtained certificate through the *$acme_cert_<name>* variable, regardless of whether the block contains an `acme` directive.

For example, if you have a wildcard server block that already covers all subdomains, additional server blocks for specific subdomains do not need the `acme` directive:

```
http {

    resolver 127.0.0.53;

    acme_client example https://acme-v02.api.letsencrypt.org/directory
        challenge=dns;
```

```
    # This block lists the domains for the certificate request
    server {

        listen 443 ssl;

        server_name example.com *.example.com;
        acme example;

        ssl_certificate $acme_cert_example;
        ssl_certificate_key $acme_cert_key_example;
    }

    # This block uses the same certificate but does not
    # add its server_name to the certificate request
    server {

        listen 443 ssl;

        server_name app.example.com;

        ssl_certificate $acme_cert_example;
        ssl_certificate_key $acme_cert_key_example;
    }
}
```

Explicit Domain List

To control the exact set of domain names in a certificate without relying on automatic collection from all server blocks, create a dedicated `server` block that contains only the *server_name* and *acme* directives. To prevent this block from handling real traffic, bind it to a Unix domain socket:

```
# Dedicated block that defines the certificate's domain list
server {

    listen unix:/tmp/acme_example.sock;

    server_name example.com www.example.com;
    acme example;
}
```

Other `server` blocks can then use the certificate through the *$acme_cert_<name>* variable without affecting which domains are requested.

Separate Certificates for Different Domains

Each *acme_client* manages a single certificate. To obtain several independent certificates (for example, for unrelated domains that should not share one certificate), configure a separate *acme_client* for each in the `http` block, and point the *acme* directive of every `server` block at the relevant client by name:

```
http {

    resolver 127.0.0.53;

    # Two independent clients, each managing its own certificate
    acme_client shop https://acme-v02.api.letsencrypt.org/directory
        challenge=http;
```

```
    acme_client blog https://acme-v02.api.letsencrypt.org/directory
        challenge=http;

    server {

        listen 443 ssl;

        server_name shop.example.com www.shop.example.com;
        acme shop;

        ssl_certificate $acme_cert_shop;
        ssl_certificate_key $acme_cert_key_shop;
    }

    server {

        listen 443 ssl;

        server_name blog.example.com;
        acme blog;

        ssl_certificate $acme_cert_blog;
        ssl_certificate_key $acme_cert_key_blog;
    }
}
```

HTTP Validation

Validation is handled automatically. The process involves the ACME server, upon receiving a request, retrieving a special token file via HTTP from the client at the address `/.well-known/acme-challenge/<TOKEN>`. Our ACME module tracks and processes such requests automatically. When the expected response with the correct content is received, the ACME server confirms that the domain belongs to the client.

Configuration Example

In this example, the ACME client named `example` manages certificates for `example.com` and `www.example.com` (note that wildcard certificates 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; # Optional if no server listens on the HTTP challenge port
                   # (see 'acme_http_port' directive)

        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 ACME calls. If no server is configured to listen on the HTTP challenge port, the module creates a dedicated listener on port 80 (or the one set in *acme_http_port*). A separate *server* block is not required.

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 that the domain belongs to the client.

Our 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.

Note

The Angie server must be reachable from the internet on UDP port 53 (or the one specified in *acme_dns_port*). If the server is behind a firewall, make sure this port is open for incoming connections.

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 `ns.example.com` is accessible by mapping it to the IP address (`93.184.215.14`).

Warning

NS record propagation can take anywhere from a few minutes to 48 hours depending on TTL and DNS provider. It is recommended to verify the configuration is correct before requesting a certificate.

To verify that DNS is configured correctly, you can use the following commands:

```
$ dig NS _acme-challenge.example.com +short  # Check NS record for _acme-challenge␣
↪subdomain

  ns.example.com.

$ dig A ns.example.com +short  # Check A record for name server

  93.184.215.14

$ nc -zv 93.184.215.14 53  # Check DNS server accessibility on port 53
```

This method allows requesting wildcard certificates, for example, a certificate that includes 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.

Warning

The applicability of this scenario largely depends on the capabilities provided by your DNS provider; some providers do not allow such configurations.

Configuration Example

Overall, the configuration is similar to the example in the previous section. There is no need for HTTP-specific settings; instead, it's sufficient to set `challenge=dns` for the *acme_client* directive.

In this example, the ACME client named `example` manages 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;
    }
}
```

ALPN Validation

Validation is handled automatically. The ACME server connects using TLS and requests the `acme-tls/1` protocol via ALPN. The module serves a temporary certificate for the validation request.

To enable this method, configure `challenge=alpn` in the *acme_client* directive and ensure your TLS listener is reachable on port 443 (or the port used for TLS).

Configuration Example

The configuration is similar to the previous sections; it is enough to set `challenge=alpn` for the *acme_client* directive and ensure the TLS server is reachable on port 443.

In this example, the ACME client named `example` manages a certificate for `example.com` and `www.example.com`:

```
http {

    resolver 127.0.0.53;

    acme_client example https://acme-v02.api.letsencrypt.org/directory
        challenge=alpn;

    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;
    }
}
```

Hook-Based Validation

Unlike the previous methods, this validation requires additional effort. The ACME server performs 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 hook calls (*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.

When certificate issuance or renewal requires domain verification, Angie generates an internal request to the named `location` containing the *acme_hook* directive. How this request is handled depends entirely on the other directives configured in the same `location`.

The general pattern is:

1. Create a named `location` with the *acme_hook* directive.
2. Configure a request handler in the same `location` using whatever module fits your setup: *fastcgi_pass* for FastCGI, *proxy_pass* for HTTP, `cgi` for CGI scripts, etc.
3. Pass ACME *variables* to the handler using the mechanism it supports, for example `fastcgi_param` for FastCGI or `cgi_set_var` for CGI.

The handler 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 handler is ignored.

Minimal Configuration

Regardless of the handler used, the hook `location` follows this structure:

```
location @acme_hook_location {

    acme_hook example;

    # Handler directive (fastcgi_pass, proxy_pass, cgi on, ...)
    # Pass ACME variables using the handler's mechanism:
    #   ACME_HOOK       - $acme_hook_name ("add" or "remove")
    #   ACME_CHALLENGE  - $acme_hook_challenge ("dns" or "http")
    #   ACME_DOMAIN     - $acme_hook_domain
    #   ACME_TOKEN      - $acme_hook_token
    #   ACME_KEYAUTH    - $acme_hook_keyauth
}
```

For DNS validation, the handler must use `ACME_HOOK` to determine the action: when it is `add`, create a TXT record for `_acme-challenge.`*`ACME_DOMAIN`* with the value from `ACME_KEYAUTH`; when it is `remove`, delete that record.

FastCGI Example

In this example, the ACME client `example` is configured for domain verification using DNS callbacks, indicated by the `challenge=dns` parameter in the *acme_client* directive.

The `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 handles the hook calls. The *acme_hook* directive associates it with the ACME client `example`. Hook requests are sent to a local FastCGI server on port 9000 using *fastcgi_pass*. The `fastcgi_param` directives pass the ACME variables 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 corresponding 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 the external DNS server to successfully pass validation, similar to the process described in *DNS Validation*. The implementation details of 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;
}
```

PHP-FPM Example

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['...']`.

ACME in the Stream Module

The *ACME* stream module enables automated certificate issuance and usage for TCP traffic. For it to work correctly, you must first configure its HTTP counterpart: the ACME client must be declared in the `http` context, and the `stream` block itself must be placed *after* the `http` block in the configuration.

Configuration Example

By default, HTTP validation mode is used to obtain certificates. As mentioned in the *HTTP Validation* section, this requires an HTTP server listening on port 80:

```
# HTTP part
http {

    resolver 127.0.0.53;

    # ACME client for the stream part
    acme_client example https://acme-v02.api.letsencrypt.org/directory;

    # Server for HTTP validation
    server {

        listen 80;
        return 444;
    }
}

# Stream part
stream {

    server {

        listen 12345 ssl;
        proxy_pass backend_upstream;

        ssl_certificate $acme_cert_example;
        ssl_certificate_key $acme_cert_key_example;

        server_name example.com www.example.com;
        acme example; # reference to the ACME client defined in the HTTP part
    }

    upstream backend_upstream {

        server 127.0.0.1:54321;
    }
}
```

You can also use DNS validation by configuring `challenge=dns` in the *acme_client* directive; in that case, the server will not be needed.

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 our ACME module.

Suppose you configured certificates as follows:

```
$ sudo certbot --nginx -d example.com -d www.example.com
```

The configuration automatically created by this command 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;
}
```

In the example above, the highlighted lines need to be modified. Depending on your circumstances and preferences, configure *HTTP validation* or *DNS validation* using the ACME module.

The resulting *Angie configuration* might look something 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;
    }
}
```

Remember to reload the configuration *after making changes*:

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

Once you have verified that this configuration works, you can delete the `certbot` certificates and disable or remove certbot entirely from the server if it is no longer used elsewhere, for example:

```
$ 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
```

3.4.2 Angie Cluster Setup

This guide describes the process of creating a fault-tolerant Angie cluster with automatic configuration synchronization and virtual IP address failover.

Preparing Cluster Nodes for Synchronization

The first step is to prepare all cluster nodes by configuring user accounts and ensuring secure access between servers.

Configuring Users and Access Permissions

Create a user on all nodes (for example, `angie-ha-sync`) with `sudo` privileges:

```
$ sudo adduser angie-ha-sync
```

Set a password if necessary:

```
$ sudo passwd angie-ha-sync
```

Note

In some operating systems (for example, Alt Linux), you should add the user to the `wheel` group:

```
$ sudo usermod -a -G wheel angie-ha-sync
```

To work with `rsync` when MAC is enabled in Astra Linux, set the correct integrity level:

```
$ sudo pdpl-user -i 63 angie-ha-sync
```

Configure sudo without password:

```
$ echo "angie-ha-sync ALL=(ALL:ALL) NOPASSWD:ALL" | sudo tee -a /etc/sudoers
```

On the master node, create SSH keys and copy them to backup nodes:

```
$ su - angie-ha-sync
$ ssh-keygen -t rsa
$ ssh-copy-id angie-ha-sync@node2_hostname
```

Warning

Before copying SSH keys, ensure that the `/etc/ssh/sshd_config` file has the option:

`PasswordAuthentication yes`

After setting up key-based access, set the value to `no` to improve security.

Note

For cross-synchronization of Angie configuration, copy the user keys to all nodes:

```
$ scp -p ~angie-ha-sync/.ssh/id_rsa angie-ha-sync@node2_hostname:.ssh/
```

Installing Angie and angie-ha-sync

After preparing the nodes, you need to install the main cluster components: Angie and the configuration synchronization package.

Configure the repository on all nodes according to the instructions for your system packages:

- Instructions for Angie
- Instructions for Angie PRO

Installing angie-ha-sync

The configuration synchronization module is available in the following packages:

- Angie: `angie-ha-sync`
- Angie PRO: `angie-pro-ha-sync`

Note

When installing this package on a clean system, the corresponding `angie` or `angie-pro` package will also be installed as a dependency.

On all nodes, install the package using your OS package manager, for example:

```
$ sudo {apk|apt|pkg|yum|zypper} {add|install} angie-pro-ha-sync
```

Configuring Configuration Synchronization

The next step is setting up automatic synchronization of configuration files between cluster nodes.

Note

Synchronization principles:

- Synchronization is performed via `rsync`.
- Only occurs when the Angie service is running.

- Executed manually (command `angiehasync -Sd`).
- Works in one direction: from master node to backup.
- `rsync` runs in daemon mode.

Configuring `rsync`

Create an `rsync` configuration (`/etc/rsyncd.conf`) on the nodes:

```
[angie] # Directory with Angie configuration
    path = /etc/angie
# User for synchronization
    uid = angie-ha-sync
# User group
    gid = angie-ha-sync
# IP or subnet from which connections are allowed
    hosts allow = 10.21.8.0/24
# Deny all others
    hosts deny = *
```

Depending on the OS, start the daemon:

```
$ sudo service rsyncd start # or $ sudo service rsync start
```

Note

For some systems, there are ready-made instructions:

- Alt
- Astra

Configuring the Synchronization File

Edit `/etc/angiehasync/angiehasync.conf`:

```
M_NODE="<node1_hostname>"             # Hostname or IP of this node
TARGET_HOSTS="<node2_hostname>"       # List of hosts/IPs for synchronization (space-
↪separated).
                                      # Can be omitted on backup nodes.
SSH_USER="user"                       # User for synchronization (with administrator␣
↪privileges)
SSH_ID="/home/$SSH_USER/.ssh/id_rsa" # Path to private key
```

Note

For cross-synchronization, fill in the `TARGET_HOSTS` list on all nodes; however, do not include the current node that is currently being configured in the list.

Configuring Health Checks for Angie

Add a health check block to the Angie configuration (`/etc/angie/angie.conf`):

```
server {
```

```
    listen unix:/tmp/angie_hcheck.sock; # Unix socket for checking
    access_log off;
    error_log /dev/null;
    default_type text/plain;
    return 200 'ok\n';
}
```

Start Angie:

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

Start synchronization:

```
$ sudo angiehasync -Sd
```

Note

The script will automatically check the configuration, perform synchronization with all nodes, and apply it.

Configuring Keepalived

For automatic failover between cluster nodes, Keepalived is used — a service for managing virtual IP addresses (VIP).

Note

If the `keepalived` package is not installed — install it:

```
$ sudo {apk|apt|pkg|yum|zypper} {add|install} keepalived
```

To bind processes to non-local IP addresses, allow the system to perform corresponding actions:

```
$ sudo sysctl -w net.ipv4.ip_nonlocal_bind=1
```

More details: https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt#ip_nonlocal_bind

Suppose VIP `10.21.11.230` is assigned either to the master node (`10.21.8.26`) or to the backup (`10.21.8.27`).

If Angie listens on this VIP (`listen 10.21.11.230:80;`) but the address is not yet assigned, Angie will not be able to start without the `ip_nonlocal_bind` parameter.

Keepalived Configuration

On the master node (`/etc/keepalived/keepalived.conf`):

```
global_defs {
    enable_script_security
}

vrrp_script angie_check {
    script "/usr/bin/curl -s --connect-timeout 5 -A 'angie_hcheck_script'
    --no-buffer -XGET --unix-socket /tmp/angie_hcheck.sock http://hcheck/"
    interval 5 user angie
}
```

```
vrrp_instance angie {
    state MASTER interface enp0s2 virtual_router_id 254 priority 100
    advert_int 2 unicast_src_ip 10.21.8.26

    unicast_peer {
        10.21.8.27
    }

    virtual_ipaddress {
        10.21.11.230
    } track_script {
        angie_check
    }
}
```

On the backup node:

```
global_defs {
    enable_script_security
}

vrrp_script angie_check {
    script "/usr/bin/curl -s --connect-timeout 5 -A 'angie_hcheck_script'
    --no-buffer -XGET --unix-socket /tmp/angie_hcheck.sock http://hcheck/"
    interval 5 user angie
}

vrrp_instance angie {
    state MASTER interface enp0s2 virtual_router_id 254 priority 99
    advert_int 2 unicast_src_ip 10.21.8.27

    unicast_peer {
        10.21.8.26
    }

    virtual_ipaddress {
        10.21.11.230
    } track_script {
        angie_check
    }
}
```

Note

In the `vrrp_instance angie` section, set the following values:

- `unicast_src_ip` — IP of the current node
- `unicast_peer` — IP of neighboring nodes
- `virtual_ipaddress` — virtual IP (VIP)
- `interface` — network interface

Start the service:

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

Keepalived Configuration Breakdown

Let's examine the main elements of the Keepalived configuration in detail to understand the cluster operation principles.

The configuration includes two parts:

- `global_defs` — global settings
- `vrrp_instance` — VRRP parameters (VIP switching)

Main elements:

- `enable_script_security` — allows execution of health check scripts
- `vrrp_script` — Angie health check script
- `state MASTER` — initial node state
- `priority` — priority (MASTER role is assigned to the highest)
- `advert_int` — VRRP advertisement interval
- `unicast_src_ip` — current node IP
- `unicast_peer` — neighbor IPs
- `virtual_ipaddress` — VIP address
- `track_script` — availability monitoring via health check scripts

Note

If the original master node recovers, it will regain the MASTER role (higher priority). To disable failback, use the `nopreempt` parameter:

```
vrrp_instance angie {
    ... nopreempt
}
```

Testing Cluster Operation

After completing the configuration, it's necessary to test the cluster operation and ensure correct switching between nodes.

Check VIP status:

```
$ ip addr show enp0s2 | grep "10.21.11.230"
```

Test fault tolerance:

Stop Angie on the master node:

```
$ sudo service angie stop
```

Check VIP transition to the backup node:

```
$ ip addr show enp0s2 | grep "10.21.11.230"
```

Start Angie on the master node again:

```
$ sudo service angie start
```

After this, the VIP should return to the master node.

3.4.3 OIDC Authentication Setup

This guide explains how to set up OpenID Connect (OIDC) authentication using Google as the identity provider and Angie web server with Lua scripting.

The implementation protects internal endpoints with OAuth2/OIDC authentication and demonstrates one way to restrict access based on email domains. This is just one example approach; you can implement access control however you prefer, such as maintaining allowlists of specific users, checking for realm membership or group attributes in the provider response, or using custom claims from your private IAM system.

Tip

This OIDC implementation provides a foundation for authentication but should be adapted for production use with proper security measures, monitoring, and compliance with your organization's security policies.

Architecture

The OIDC setup suggested here consists of:

- Angie - with Lua module support for OIDC processing
- lua-resty-openidc - OpenResty Lua library for OIDC authentication
- Google OAuth2 - Identity provider for user authentication
- Docker Compose - Used in this example for rapid startup only; use whatever deployment approach you prefer in production

Prerequisites

Before configuring OIDC authentication, ensure you have:

1. Angie web server with Lua module support
2. Docker and Docker Compose (for deployment)
3. A Google Cloud Console project
4. OAuth2 credentials from Google

Google OAuth2 Setup

To configure Google as your OIDC provider:

1. Navigate to the Google Cloud Console
2. Create a new project or select an existing one
3. Configure the OAuth consent screen for your project (External or Internal) and publish it so users can authenticate
4. Create OAuth2 credentials:
    - Application type: Web application
    - Authorized redirect URIs: `http://localhost/auth/callback`
5. Save your `client_id` and `client_secret` for configuration

Note

Standard Google Identity Services already support OIDC; the legacy Google+ API is not required. Enable additional Google APIs only if your application needs their data.

Configuration Setup

Let's start with the configuration files needed for the OIDC setup.

Docker Compose Configuration

The Docker deployment uses the following configuration file:

Listing 1: docker-compose.yml

```
services:
  angie:
    image: docker.angie.software/angie:templated
    environment:
      ANGIE_LOAD_MODULES: "lua"
    ports:
      - 80:80
    volumes:
      - ./files/etc/angie/http.d:/etc/angie/http.d
```

This configuration does the following:

- Uses the templated Angie image with Lua module support
- Loads the Lua module for OIDC functionality
- Maps port 80 for HTTP access
- Mounts configuration files from local directory

For a plug-and-play demo, download the `OIDC quick-start bundle`, set your `client_id` and `client_secret` in `files/etc/angie/http.d/oidc.lua`, and everything will work out of the box.

OIDC Authentication Script

Create an OIDC authentication script that handles the authentication logic using the `lua-resty-openidc` library:

Listing 2: /etc/angie/http.d/oidc.lua

```
access_by_lua_block {
    local res, err = require("resty.openidc").authenticate({
        redirect_uri = "http://localhost/auth/callback",
        discovery = "https://accounts.google.com/.well-known/openid-configuration",
        logout_path = "/auth/logout",
        redirect_after_logout_uri = "/auth/logged-out",
        revoke_tokens_on_logout = true,
        client_id = "YOUR_CLIENT_ID",
        client_secret = "YOUR_CLIENT_SECRET"
    })
}
```

Configuration parameters:

- `redirect_uri`: Callback URL after successful authentication
- `discovery`: Google's OIDC discovery endpoint
- `logout_path`: Path for user logout
- `redirect_after_logout_uri`: Redirect destination after logout
- `revoke_tokens_on_logout`: Revoke tokens on logout for security
- `client_id` and `client_secret`: Your Google OAuth2 credentials

Angie Configuration

Configure Angie with the necessary *location* blocks for OIDC authentication.

Protected Resources

To protect resources with OIDC authentication:

```
location /internal/ {
    include /etc/angie/http.d/oidc.lua;
    proxy_pass http://127.0.0.1/status/;
}
```

This configuration does the following:

- Protects the `/internal/` path with OIDC authentication
- Proxies authenticated requests to the *internal API* at `/status/`

Authentication Endpoints

Configure the OAuth2 flow endpoints:

```
location /auth/callback {
    include /etc/angie/http.d/oidc.lua;
}

location /auth/logout {
    include /etc/angie/http.d/oidc.lua;
}

location /auth/logged-out {
    default_type text/plain;
    return 200 "You have been logged out. Bye!";
}
```

Endpoint functions:

- `/auth/callback`: Handles OAuth2 callback from Google
- `/auth/logout`: Initiates user logout
- `/auth/logged-out`: Landing page after successful logout

Internal API Access

Configure restricted access to the internal API:

```
location /status/ {
    api     /status/;
    allow   127.0.0.1;
    deny    all;
}
```

This provides:

- Access to Angie's *status API*
- Localhost-only access (127.0.0.1)
- OIDC protection when accessed through `/internal/`

Deployment Steps

Follow these steps to deploy OIDC authentication:

Configuration Update

1. Update OAuth2 credentials in your OIDC Lua script:

   Replace the placeholder values in `oidc.lua`:

   - `client_id` with your Google OAuth2 client ID
   - `client_secret` with your Google OAuth2 client secret

Service Startup

1. Start the Docker services:

   ```
   $ docker-compose up -d
   ```

2. Verify the deployment:

   - Navigate to `http://localhost/internal/`
   - You should be redirected to Google for authentication
   - After successful login, you'll access the protected content

Security Configuration

Email Domain Restriction

Implement domain validation to restrict access by email domain:

```
if not string.match(res.user.email, "gmail.com$") then
    ngx.exit(ngx.HTTP_FORBIDDEN)
end
```

For production environments, consider the following:

- Replacing `gmail.com` with your organization's domain
- Implementing a whitelist of allowed email addresses
- Adding role-based access control

Token Management

The OIDC implementation includes security features:

- Tokens are automatically revoked on logout (`revoke_tokens_on_logout = true`)
- Sessions are managed securely by the lua-resty-openidc library
- All authentication flows follow OAuth2/OIDC security best practices

Warning

For production deployments:

- Always use HTTPS for OAuth2 callbacks
- Store client secrets securely, never in version control
- Implement proper session timeouts and renewal policies
- Monitor authentication logs for security events

Authentication Flow

The OIDC authentication flow follows standard OAuth2/OIDC procedures:

1. User accesses a protected URL (e.g., `http://localhost/internal/`)
2. If not authenticated, user is redirected to Google OAuth2
3. User authenticates with Google credentials
4. Google redirects back to `/auth/callback` with authorization code
5. Server exchanges code for access token and ID token
6. User information is validated (including email domain check)
7. User is granted access to protected resource

The logout process ensures secure session termination:

1. User navigates to `http://localhost/auth/logout`
2. Tokens are revoked at Google's OAuth2 endpoint
3. Local session data is cleared
4. User is redirected to `/auth/logged-out`

Advanced Configuration

To restrict access to a specific organization domain:

```
if not string.match(res.user.email, "yourcompany.com$") then
    ngx.exit(ngx.HTTP_FORBIDDEN)
end
```

For organizations with multiple allowed domains:

```
local allowed_domains = {"company1.com", "company2.com", "gmail.com"}
local email_valid = false

for _, domain in ipairs(allowed_domains) do
    if string.match(res.user.email, domain .. "$") then
        email_valid = true
        break
    end
end

if not email_valid then
    ngx.exit(ngx.HTTP_FORBIDDEN)
end
```

User Information Access

Access additional user claims from the OIDC provider:

```
-- Access user information from ID token
local user_name = res.user.name
local user_picture = res.user.picture
local user_locale = res.user.locale
local user_email = res.user.email
```

These values can be used for logging, personalization, or additional access control decisions.

3.4.4 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;
    #...
```

SSL 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, for example:

```
$ 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 Angie 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

3.4.5 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
1.8.2	23.01.2026	Fixed the link to Angie ADC documentation.
1.8.1	08.09.2025	Fixed incorrect terms in Settings and tooltips.
1.8.0	03.07.2025	Display of response time metrics for proxied HTTP and TCP/UDP servers
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	• Internationalization 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;
}
```

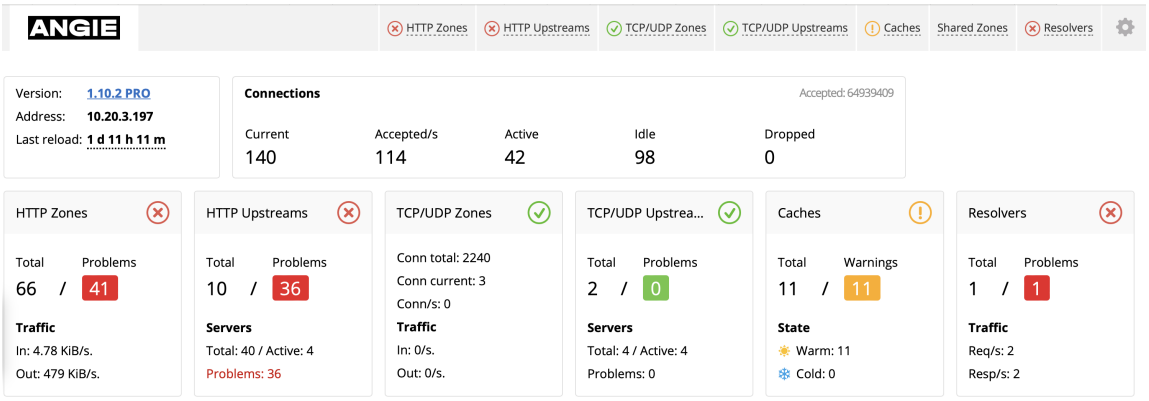
Interface

The console is a single screen with a set of tabs, each containing several widgets with monitoring data.

Tip

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.

Note

Statistics widgets are displayed if the corresponding blocks are configured in the *Angie configuration*.

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

Warning

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

Warning

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

Warning

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

Warning

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

Warning

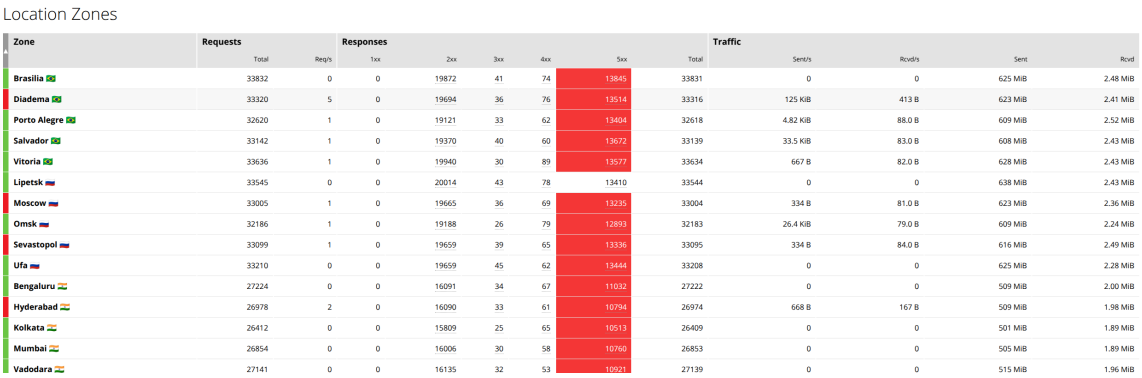
Requires setting the *status_zone* directive in a `server` or `location` context.

Server Zones Section

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 **Tip** 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	33832	0	0	19872	41	74	13845	33831	0	0	625 MiB	2.48 MiB
Diadema	33320	5	0	19694	36	76	13514	33316	125 KiB	413 B	623 MiB	2.41 MiB
Porto Alegre	32620	1	0	19121	33	62	13404	32618	4.82 KiB	88.0 B	609 MiB	2.52 MiB
Salvador	33142	1	0	19370	40	60	13672	33139	33.5 KiB	83.0 B	608 MiB	2.43 MiB
Vitoria	33636	1	0	19940	30	89	13577	33634	667 B	82.0 B	628 MiB	2.43 MiB
Lipetsk	33545	0	0	20014	43	78	13410	33544	0	0	638 MiB	2.43 MiB
Moscow	33005	1	0	19665	36	69	13235	33004	334 B	81.0 B	623 MiB	2.36 MiB
Omsk	32186	1	0	19188	26	79	12893	32183	26.4 KiB	79.0 B	609 MiB	2.24 MiB
Sevastopol	33099	1	0	19659	39	65	13336	33095	334 B	84.0 B	616 MiB	2.49 MiB
Ufa	33210	0	0	19659	45	62	13444	33208	0	0	625 MiB	2.28 MiB
Bengaluru	27224	0	0	16091	34	67	11032	27222	0	0	509 MiB	2.00 MiB
Hyderabad	26978	2	0	16090	33	61	10794	26974	668 B	167 B	509 MiB	1.98 MiB
Kolkata	26412	0	0	15809	25	65	10513	26409	0	0	501 MiB	1.89 MiB
Mumbai	26854	0	0	16006	30	58	10760	26853	0	0	505 MiB	1.89 MiB
Vadodara	27141	0	0	16135	32	53	10921	27139	0	0	515 MiB	1.96 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 **Tip** 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

Connection Limit Zones (Limit Conn) Section


Limit Conn

Zone	Passed	Rejected	Exhausted	Skipped
limit-conn-http	163972	2802	0	0

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:

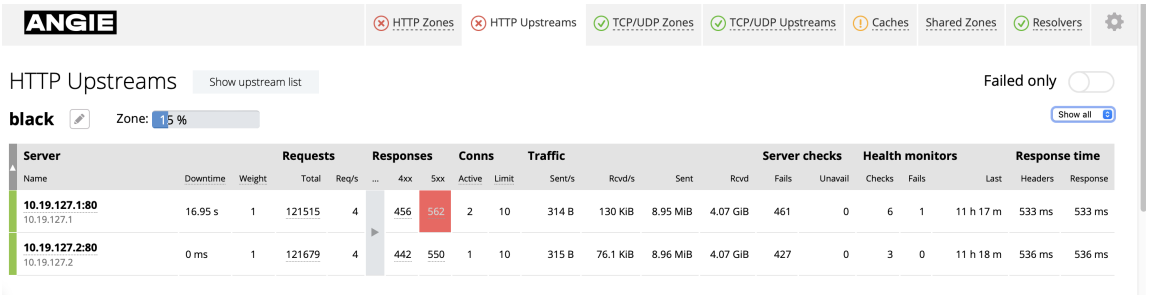
Zone	Zone name **Tip** Click the icon next to *Zone* to open or close the chart with the following indicators.
Passed	Total number of proxied connections
Rejected	Total number of rejected connections
Exhausted	Total number of connections dropped due to zone storage overflow
Skipped	Total number of connections passed with a zero or greater than 255 bytes key

Request Limit Zones (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 **Tip** 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



Warning

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. In debug mode, memory usage percentage is also displayed.

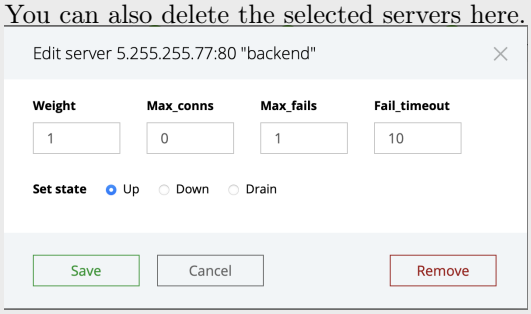
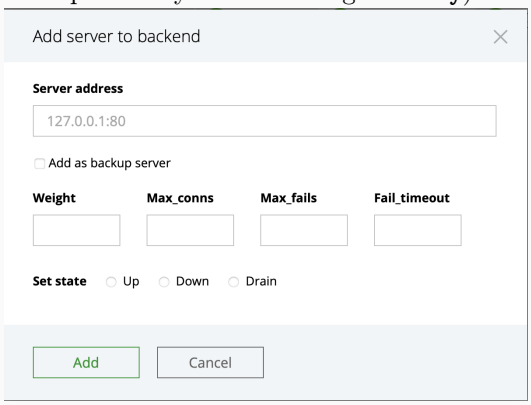
- The *Show upstreams list* button toggles a brief list of upstreams with the number of problematic upstreams and peers.
- The *Failed only* switch toggles the display mode for problematic upstreams statistics.
- The edit button toggles the *upstream editing* interface.
- The dropdown list on the right side of each upstream table allows you to filter servers in a specific state (*Up*, *Failed*, *Checking*, *Down*).

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 **Tip** 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
Response time	Time from the beginning of the request to sending the first byte of the response; total time from the beginning of the request to completion of sending the entire response (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`). 

`TCP/UDP Zones` Tab

Warning

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:

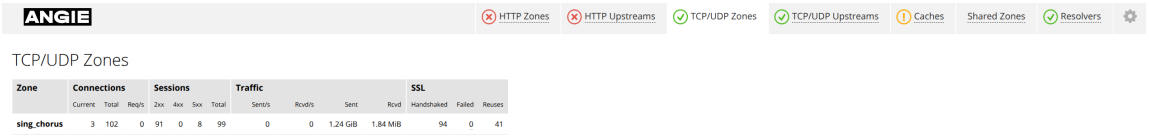
```
stream {

    # ...
    limit_conn_zone $connection zone=limit-conn-stream:10m;
```

```
    server {

        # ...
        limit_conn limit-conn-stream 1;
        status_zone foo;
    }
}
```

`TCP/UDP Zones` Section



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:

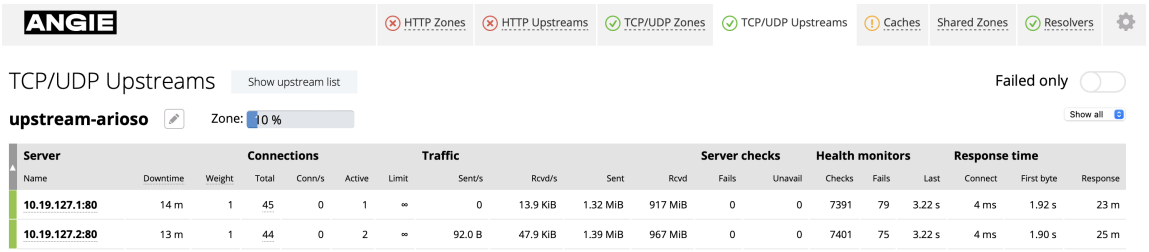
`Zone`	Zone name
`Connections`	Current and total number of connections, as well as the number of connections per second
`Sessions`	Number of sessions broken down by status codes, as well as their total number
`Traffic`	Outgoing and incoming traffic rates, as well as total volumes of outgoing and incoming traffic
`SSL`	Aggregate counts of: successful SSL handshakes; unsuccessful SSL handshakes; SSL session reuses

`Connection Limit Zones (Limit Conn)` Section

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 **Tip** 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



Warning

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. In debug mode, memory usage percentage is also displayed.

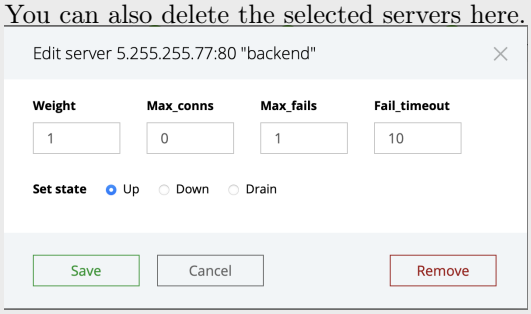
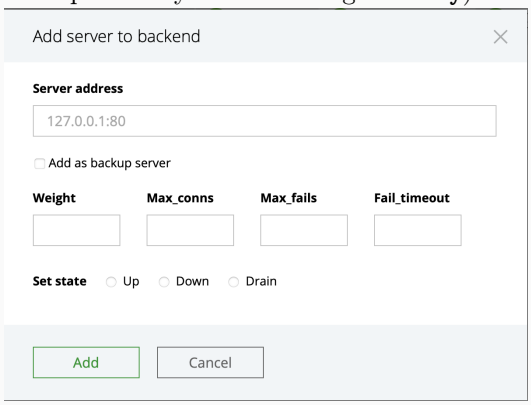
- The `Show upstreams list` button toggles the display of a brief list of upstreams with the number of problematic upstreams and peers.
- The `Failed only` switch enables and disables the display mode for problematic upstreams statistics.
- The edit button opens the `upstream editing` widget.
- The dropdown list on the right side of each upstream table allows you to filter servers in a specific state (`Up`, `Failed`, `Checking`, `Down`).

For each upstream, the following data is presented:

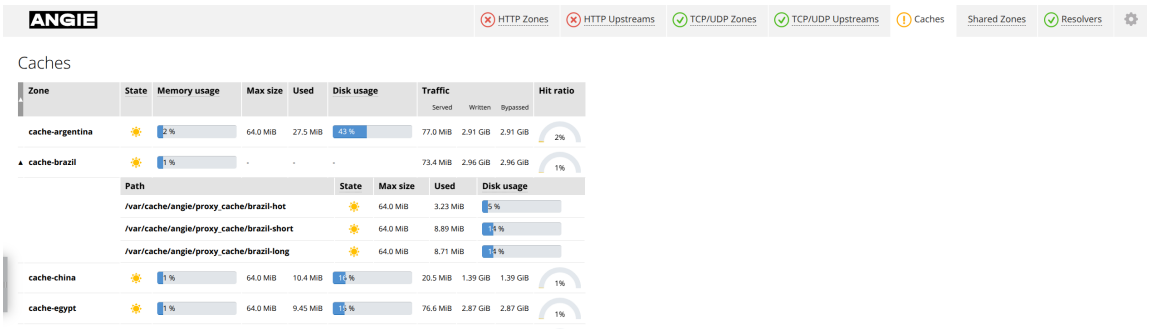
`Server`	Names, downtimes, and weights of upstream servers **Tip** Click the arrow next to `Server` to sort servers by their state or configuration order.
`Connections`	Number of active connections and their maximum limit, if set
`Traffic`	Outgoing and incoming traffic rates, as well as total volumes 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)
`Health monitors`	Total number of server checks, number of unsuccessful checks, and the time of the last check
`Response time`	Time spent establishing a connection to the backend; time from the beginning of the request to receiving the first byte of the response; total time elapsed from the beginning of the request to receiving the last byte of the response (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



Warning

Requires setting the *proxy_cache_path* directive in the `http` context.

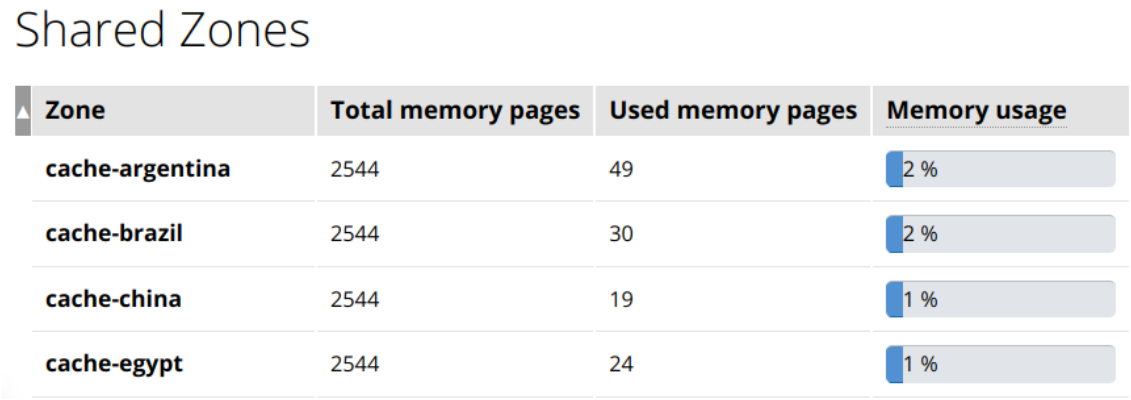
This tab summarizes monitoring statistics for `proxy_cache` zones in the `http` context, generated from the */status/http/caches/* API section. The following data is presented for each zone:

`Zone`	Zone name **Tip** 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 **Tip** 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

`DNS Resolvers` Tab

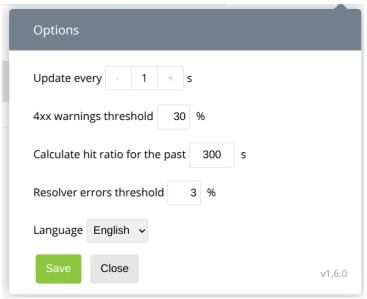
Warning

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 **Tip** 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.

3.4.6 Custom Metrics Configuration

Angie can collect custom numeric metrics in shared memory and expose them via the real-time *statistics API* at `/status/http/metric_zones/`. This is provided by the *Metric* module.

Configuration Steps

1. Define a metric zone in the `http` block:
   - *metric_zone* creates a zone with a single metric mode.
   - *metric_complex_zone* creates a zone with multiple named metrics.
2. Update metrics in request processing with the *metric* directive. Use a `key=value` pair (both are *complex values*), and choose the update stage with `on=` (`request`, `response`, or `end`).
3. Expose the API with a `location`:

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

Example

Count requests per host and expose the metrics in the API:

```
http {
    metric_zone requests:128k count;

    server {
        listen 80;

        location / {
            metric requests $host=1;
        }

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

Notes

- If `expire=on` is set on the zone and the shared memory is full, the least recently used entries are expired. If `expire=off`, new updates are discarded and the `discarded` counter grows.
- If `discard_key` is set, metrics from expired entries are aggregated under that key in the API output.
- Keys and values are limited to 255 bytes; longer keys are truncated in the API.
- An empty value is treated as `0`, and a non-empty value without a leading number is treated as `1`.

3.4.7 Migrating from nginx to Angie

If you're switching from nginx to Angie, congratulations! We have a guide for you.

Keep in mind that it's tailored for a basic replacement scenario that relies on a packaged version of Angie. If you're working with containers, virtual machines, custom paths, or modules, you'll need additional adjustments.

Installing Angie

We recommend using the official packages from our repositories; see the installation steps for Angie for your distribution. Don't start the server yet; instead, check it with the command `sudo angie -V`:

```
$ sudo angie -V

  Angie version: Angie/1.11.8
  nginx version: nginx/1.29.3
  built by gcc 11.4.0
  configure arguments: --prefix=/etc/angie --conf-path=/etc/angie/angie.conf ...
```

As this shows, the *configuration* is located in `/etc/angie/` when Angie is installed from a package.

Updating Angie Configuration

Angie usually requires minimal changes to existing nginx configuration.

1. Copy the entire nginx configuration to `/etc/angie/`:

   ```
   $ sudo rsync -a --no-links /etc/nginx/ /etc/angie/
   ```

   We assume nginx configuration is stored in `/etc/nginx/`; adjust the steps if you have a different path.

2. Rename the main configuration file as Angie expects:

   ```
   $ sudo mv /etc/angie/nginx.conf /etc/angie/angie.conf
   ```

3. Update paths throughout the Angie configuration, starting with the main configuration file. Details depend on how nginx was installed, but at minimum you need to update the following.

   Any *include* paths that still point to `/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 important for Angie process management:

```
# pid /var/run/nginx.pid;
# -- or --
# pid /run/nginx.pid;
pid /run/angie.pid;
```

Finally, *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 the `sites-enabled/` directory is used to include virtual hosts, update it as well:

```
# include /etc/nginx/sites-enabled/*;
include /etc/angie/sites-enabled/*;
```

Then recreate the symlinks in `/etc/angie/sites-enabled/` to make everything 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 to `/etc/angie/` earlier, 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 equivalents for 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 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;
```

Then copy the module configuration files:

```
$ sudo rsync -a /usr/share/nginx/modules/ /usr/share/angie/modules/
```

Finally, change 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 dynamic modules are included via `/etc/nginx/modules-enabled/`, update the path:

```
# include /etc/nginx/modules-enabled/*.conf;
include /etc/angie/modules-enabled/*.conf;
```

Then recreate the symlinks in `/etc/angie/modules-enabled/` to make everything 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 to `/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 *root* points to the `/usr/share/nginx/html/` directory, you can change the directive to point to Angie.

Copy the directory and update the `root` value 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 sufficient to leave the *user* directive as is, you can use Angie accounts for flexibility.

Update the `user` settings in Angie configuration:

```
# user www-data www-data;
user angie angie;
```

Change the owner of *all* configuration files, including files in `/usr/share/angie/`, for example:

```
$ sudo chown -R angie:angie /etc/angie/
$ sudo chown -R angie:angie /usr/share/angie/
```

If the Angie configuration has *root* directives, change the owner of the directories specified there, for example:

```
$ sudo chown -R angie:angie /var/www/html/
```

Wrapping Up

To make sure nothing is missed, find and fix remaining mentions of `nginx` in Angie configuration:

```
$ grep -rn --include='*.conf' 'nginx' /etc/angie/
```

Testing and Switching

After updating Angie configuration, the next step is to check its syntax to ensure Angie can work with it, and then switch over. Verify that Angie accepts the new configuration:

```
$ sudo angie -t
```

This command parses the configuration and reports errors that would block Angie startup; fix any issues and re-run the command.

Stopping nginx, Starting Angie

To minimize downtime, start Angie immediately after stopping nginx:

```
$ sudo systemctl stop nginx && sudo systemctl start angie
```

If needed, enable the Angie service to start after reboot:

```
$ sudo systemctl enable angie
```

Migration complete! That's it; you're awesome.

Disabling nginx

After confirming that Angie is running stably, you can disable or remove nginx to avoid conflicts.

The minimum you can do is disable the service:

```
$ sudo systemctl disable nginx
```

Working with SSL Certificates

If you used Certbot to manage SSL certificates with nginx, it will continue to work with Angie.

Using Certbot with Angie

Supporting Angie in Certbot requires minimal effort, since Angie is backward compatible with nginx. For Certbot to work, it's sufficient to create a symlink and specify the appropriate parameters:

```
# Create a symlink for Certbot compatibility
$ sudo ln -s /etc/angie/angie.conf /etc/angie/nginx.conf

# Obtain a certificate for a domain
$ sudo certbot --nginx --nginx-server-root=/etc/angie --nginx-ctl=angie -d example.
↪com -d www.example.com

# Automatically renew certificates
$ sudo certbot renew

# Check certificate status
$ sudo certbot certificates
```

After migrating to Angie, Certbot will continue to automatically renew certificates through configured `cron` jobs or `systemd` timers.

Migrating from Certbot to the Built-in ACME Module

Angie includes a built-in *ACME module*, which allows you to automatically obtain and renew SSL certificates without using external tools like Certbot.

Advantages of the built-in ACME module:

- full integration with Angie configuration;
- automatic certificate renewal without additional services;
- support for HTTP and DNS validation;
- ability to obtain wildcard certificates.

For detailed instructions on migrating from Certbot to the built-in ACME module, see the *Migrating from certbot* section.

Adapting Diverging Directives

A few nginx directives behave differently in Angie: some are renamed, some are deprecated, and a couple are omitted entirely. If your configuration relies on any of them, see *Unsupported nginx Directives*.

In particular, nginx's `keepalive_min_timeout` is named *lingering_timeout* in Angie; rename it if it appears in your configuration.

Configuring Angie Features

It's safe to assume you're migrating for a reason. Why not go further and configure some of the additional features available in Angie and Angie PRO that aren't in nginx?

3.4.8 Unsupported nginx Directives

Most nginx directives work in Angie unchanged, so an existing configuration usually needs no adjustment. This page lists the exceptions, the directives that Angie removes, renames, or deprecates, so you can adapt your configuration when moving from nginx.

Low-level tuning directives that nginx itself leaves undocumented, such as event-method and gzip tuning knobs, keep working in Angie with their nginx defaults and are intentionally omitted here.

Note

For the end-to-end migration process, see the *migration guide*.

Removed or Omitted

These nginx directives have no Angie equivalent.

nginx directive	Notes
`add_header_inherit`, `add_trailer_inherit`	Omitted because of their poor design; see oss_changes.

Renamed

nginx directive	Angie directive
`keepalive_min_timeout`	*lingering_timeout*

Deprecated

These directives still work but emit a warning; use the modern directive instead.

nginx directive	Use instead
`http2_idle_timeout`	*keepalive_timeout*
`http2_max_requests`	*keepalive_requests*
`http2_recv_timeout`	*client_header_timeout*
`http2_max_field_size`	*large_client_header_buffers*
`http2_max_header_size`	*large_client_header_buffers*
`proxy_downstream_buffer` (stream)	*proxy_buffer_size*
`proxy_upstream_buffer` (stream)	*proxy_buffer_size*

3.4.9 Configuring the Grafana dashboard

To configure the dashboard for Angie in Grafana, follow these steps:

1. Using the *Prometheus* module, add the following *include* directive in the `http` block of the *configuration file*:

```
http {
    include prometheus_all.conf;

    # ...
}
```

   Also add the corresponding *prometheus* directive inside a `location` within a separate `server` block with a dedicated IP address and port for this purpose, for example:

```
server {

    listen 192.168.1.100:80;

    location =/p8s {
        prometheus all;
    }
```

```
    # ...

}
```

These enable the export of Angie metrics in Prometheus format at the endpoint specified in the `location`.

2. Add the following configuration to Prometheus, specifying the IP address and port set earlier in the `server`:

```
scrape_configs:
  - job_name: "angie"
    scrape_interval: 15s
    metrics_path: "/p8s"
    static_configs:
      - targets: ["192.168.1.100:80"]
```

This will collect metrics every 15 seconds, using the `/p8s` path configured in the previous step.

> **Note**
>
> Make sure the global `scrape_interval` value does not exceed the value specified here.

3. Import the dashboard for Angie into Grafana.

3.5 Community Materials

We have collected community resources that will help you better understand configuring and using Angie.

3.5.1 Articles

- Angie: A New NGINX Fork Developed by Some of Its Former Devs on Linuxiac
- What's New in the Angie 1.9 Web Server (an nginx fork) and What to Expect from 1.10? on Habr

3.5.2 Courses

English-language courses on Angie are not currently available. Refer to the official Angie documentation and the practical guides below.

3.5.3 Practical Guides

- Migrating from Nginx to Angie: A Real-World Journey from Certbot to Built-in ACME on DEV Community

3.5.4 Interviews and Podcasts

- "NGINX is Dead? // Angie Web Server Migration Guide" by DevOps Toolbox (YouTube, 27.03.2026)
- "Nginx Has a BIG Problem..." by DevOps Toolbox (YouTube, 30.01.2026)

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 technical support.

To do this, run Angie using the executable with debug support:

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
```

Docker

In templated Docker images, you can switch to the debug version by overriding the `ANGIE_BINARY` environment variable:

```
$ docker run -it --rm -e ANGIE_BINARY="angie-debug" \
  docker.angie.software/angie:templated
```

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 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
```

In templated Docker images with debug logging enabled, you can also use the `ANGIE_ERROR_LOG_SEVERITY` environment variable:

```
$ docker run -it --rm -e ANGIE_BINARY="angie-debug" \
-e ANGIE_ERROR_LOG_SEVERITY="debug" \
docker.angie.software/angie:templated
```

If you switch to the executable without debug support 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 Directive Location

The location of the *error_log* directive affects the completeness of debug information collected.

A directive specified at a lower configuration level (for example, inside a `server` or `location` block) overrides logging settings specified at a higher level (for example, at the main configuration level or inside an `http` block).

Debug log disabled for a specific server

If debug logging is enabled globally but *error_log* is specified for an individual server without the `debug` level, debug information will not be collected for that server.

```
error_log /var/log/angie/error.log debug; # Global debug log

http {

    server {

        listen 80;
        server_name example.com;

        error_log /var/log/angie/example.com.error.log;
        # Debug log for example.com is disabled, file contains info level

        # ...
    }

    server {

        listen 80;
        server_name another.com;

        # This server will use the global debug log
        # ...
    }
}
```

Preserving debug log at server level

To preserve debug information collection for a specific server but direct it to a different file, you must also specify the `debug` level:

```
error_log /var/log/angie/error.log debug; # Global debug log

http {

    server {

        listen 80;
        server_name example.com;

        error_log /var/log/angie/example.com.error.log debug;
        # Debug log for example.com is enabled but written to a separate file

        # ...
    }
}
```

Therefore, to enable debug logging globally but override the log file for individual blocks, also specify the `debug` level in those overrides. Otherwise, if no logging level is specified in the *error_log* directive, the `error` level will be used by default and debug information for those blocks will be lost.

4.1.2 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;
}
```

4.1.3 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
```

4.2 Core Dumps

Core dumps help investigate crashes. Include them when *contacting support*. For builds from our repositories, we provide debug symbols in special packages. They have the same names as the original packages with the `-dbg` suffix added, for example `angie-dbg`.

> **Note**
>
> This section assumes you are running Angie as the `root` user (recommended).

4.2.1 Linux: systemd

To enable core dump saving when running Angie as a `systemd` service (for example, when installed from packages), modify the service settings in the `/lib/systemd/system/angie.service` file:

```
[Service]
...
LimitCORE=infinity
LimitNOFILE=65535
```

Or update the global settings in the `/etc/systemd/system.conf` file:

```
[Manager]
...
DefaultLimitCORE=infinity
DefaultLimitNOFILE=65535
```

Then reload the service configuration and restart Angie to reproduce the crash conditions:

```
$ sudo systemctl daemon-reload
$ sudo systemctl restart angie.service
```

After the crash, find the core dump file:

```
$ sudo coredumpctl -1 # optional

   TIME                            PID   UID   GID SIG COREFILE  EXE
   --- 2026-06-18 11:05:40 GMT   1157     0     0  11 present   /usr/sbin/angie

$ sudo ls -al /var/lib/systemd/coredump/  # default, see also /etc/systemd/coredump.
↪conf and /etc/systemd/coredump.conf.d/*.conf

  ...
  -rw-r----- 1 root root 177662 Jul 27 11:05 core.angie.0.
↪6135489c850b4fb4a74795ebbc1e382a.1157.1590577472000000.lz4
```

4.2.2 Linux: Manual Configuration

Check the core dump settings in the `/etc/security/limits.conf` file, modify them if necessary:

```
root soft core 0          # by default disables core dumps
root hard core unlimited  # allows increasing the size limit
```

Then increase the core dump size limit using ulimit, then restart Angie to reproduce the crash conditions:

```
$ sudo ulimit -c unlimited
$ sudo cd <path to Angie installation directory>
$ sudo sbin/angie  # or sbin/angie-debug
```

After the crash, find the core dump file:

```
$ sudo ls -al <path to Angie working directory>  # default, see /proc/sys/kernel/core_
↪pattern
  ...
  -rw-r----- 1 root root 177662 Jul 27 11:05 core.1157
```

4.2.3 FreeBSD

Check the core dump settings in the `/etc/sysctl.conf` file, modify them if necessary:

```
kern.coredump=1                                # should be 1
kern.corefile=/path/to/core/files/%N.core    # needs correct path
```

Or update the settings at runtime:

```
$ sudo sysctl kern.coredump=1
$ sudo sysctl kern.corefile=/path/to/core/files/%N.core
```

Then restart Angie to reproduce the crash conditions. If Angie is installed as a service:

```
$ sudo service angie restart
```

If Angie is installed manually:

```
$ sudo cd <path to Angie installation directory>
$ sudo sbin/angie
```

After the crash, find the core dump file:

```
$ sudo ls -al <path to core dump files>

  ...
  -rw------- 1 root root 9912320 Jul 27 11:05 angie.core
```

CHAPTER 5

Intellectual Property Rights

The documentation for the Angie PRO software product is the intellectual property of Web Server, LLC. The documentation was created as a result of modification (revision) of the documentation for the nginx software product.

Index

F

G

H

I

K

L

M

O

P

Q

R

S

T

U

V

W

X

Z