Additional modules

The first half of this chapter covered two of the most important Nginx modules: the Rewrite module and the SSI module. There are a lot more modules that will greatly enrich the functionality of the web server; they are regrouped here, thematically.

Among the modules described in this section, some are included in the default Nginx build, but some are not. This implies that unless you specifically configured your Nginx build to include these modules (as described in Chapter 1, **Downloading and Installing Nginx**), they will not be available to you. But remember that rebuilding Nginx to include additional modules is a relatively quick and easy process.

Website access and logging

The following set of modules allows you to configure the way visitors access your website and the way your server logs requests.

Index

The Index module provides a simple directive named `index`, which lets you define the page that Nginx will serve by default if no filename is specified in the client request (in other words, it defines the website index page). You may specify multiple filenames; the first file to be found will be served. If none of the specified files are found, Nginx will either attempt to generate an automatic index of the files (if the `autoindex` directive is enabled—check the HTTP Autoindex module), or return a `403 Forbidden` error page.

Optionally, you may insert an absolute filename (such as `/page.html`), but only as the last argument of the directive.

Syntax: `index file1 [file2…] [absolute_file];`

Default value: `index.html`

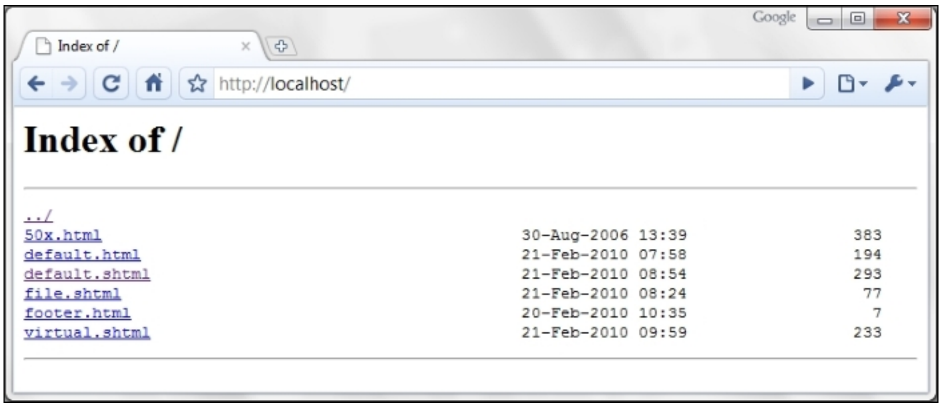
Copy

```
index index.php index.html index.htm;
index index.php index2.php /catchall.php;
```

This directive is valid in the following contexts: `http`, `server`, and `location`.

Autoindex

If Nginx cannot provide an index page for the requested directory, the default behavior is to return a `403 Forbidden HTTP` error page. With the following set of directives, you enable an automatic listing of the files that are present in the requested directory:



Index of /

```
../
50x.html                    30-Aug-2006 13:39     383
default.html                21-Feb-2010 07:58     194
default.shtml               21-Feb-2010 08:54     293
file.shtml                  21-Feb-2010 08:24      77
footer.html                 20-Feb-2010 10:35       7
virtual.shtml               21-Feb-2010 09:59     233
```

Three columns of information appear for each file—the filename, the file date and time, and the file size in bytes.

Directive	Description
`autoindex` Context: `http`, `server`, `location`	Enables or disables the automatic directory listing for directories missing an index page. Syntax: `on` or `off`

Directive	Description
`autoindex_exact_size` Context: `http` , `server` , `location`	If set to `on` , this directive ensures that the listing displays the file sizes in bytes. Otherwise, another unit is employed, such as KB, MB, or GB. Syntax: `on` or `off` Default value: `on`
`autoindex_localtime` Context: `http` , `server` , `location`	By default, this directive is set to `off` , so the date and time of files in the listing appears as the GMT time. Set it to `on` to make use of the local server time. Syntax: `on` or `off` Default value: `off`
`autoindex_format` Context: `http` , `server` , `location`	Nginx offers to serve the directory index in different formats: HTML, XML, JSON, or JSONP (by default, HTML is used). Syntax: `autoindex_format html | xml | json | jsonp;` If you set the directive value to `jsonp` , Nginx inserts the value of the `callback` query argument as JSONP callback. For example, your script should call the following URI: `/folder/?callback=MyCallbackName` .

Random index

This module enables a simple directive, `random_index` , which can be used within a `location` block for Nginx to return an index page selected randomly among the files of the specified directory.

Note

This module is not included in the default Nginx build.

Syntax: `on` or `off`

Log

This module controls the behavior of Nginx regarding the access logs. It is a key module for system administrators, as it allows analyzing the runtime behavior of web applications. It is composed of three essential directives:

Directive	Description
`access_log` Context: `http` , `server` , `location` , `if` (in location), `limit_except`	This parameter defines the access log file path, the format of entries in the access log by selecting a template name, or disables access logging. Syntax: `access_log path [format [buffer=size]] | off;` Some remarks concerning the directive syntax are as follows: • Use `access_log off` to disable access logging at the current level • The format argument corresponds to a template declared with the `log_format` directive, described next • If the format argument is not specified, the default format is employed (combined) • You may use variables in the file path

Directive	Description

`log_format`
Context: `http` , `server` , `location`
	Defines a template to be utilized by the `access_log` directive, describing the contents that should be included in an entry of the access log.
	Syntax: `log_format template_name format_string;`
	The default template is called `combined,` and matches the following example:
	Copy
	`log_format combined '$remote_addr - $remote_user [$time_local] '"$request" $status $body_bytes_sent '"$http_referer" "$http_user_agent"';`
	`# Other example`
	`log_format simple '$remote_addr $request';`

`open_log_file_cache` Context: `http` , `server` , `location`	Configures the cache for log file descriptors. Please refer to the `open_file_cache` directive of the HTTP Core module for additional information. Syntax: `open_log_file_cache max=N [inactive=time] [min_uses=N] [valid=time] | off;` The arguments are similar to the `open_file_cache` and other related directives; the difference is that this applies to access log files only.

The Log module also enables several new variables, though they are only accessible when writing log entries:

- `$connection` : The connection number
- `$pipe` : The variable is set to " `p` " if the request was pipelined
- `$time_local` : Local time (at the time of writing the log entry)
- `$msec` : Local time (at the time of writing the log entry) to the microsecond
- `$request_time` : Total length of the request processing, in milliseconds
- `$status` : Response status code
- `$bytes_sent` : Total number of bytes sent to the client
- `$body_bytes_sent` : Number of bytes sent to the client for the response body
- `$apache_bytes_sent` : Similar to `$body_bytes` , which corresponds to the `%B` parameter of Apache's `mod_log_config`
- `$request_length` : Length of the request body

Limits and restrictions

The following modules allow you to regulate access to the documents of your websites—require users to authenticate, match a set of rules, or simply restrict the access to certain visitors.

Auth_basic module

The `auth_basic` module enables the basic authentication functionality. With the two directives that it brings forth, you can make it such that a specific location of your website (or your server) is restricted to users who authenticate with a username and password:

Copy

```
location /admin/ {
    auth_basic "Admin control panel"; # variables are supported
    auth_basic_user_file access/password_file;
}
```

The first directive, `auth_basic` , can be set to either `off` or a text message, usually referred to as **authentication challenge** or **authentication realm**. This message is displayed by the web browsers in a username/password box when a client attempts to access the protected resource.

The second one, `auth_basic_user_file` , defines the path of the password file relative to the directory of the configuration file. A password file is formed of lines respecting the following syntax: `username:[{SCHEME}]password[:comment]` . Where:

- `username` : a plain text user name
- `{SCHEME}` : optionally, the password hashing method. There are currently three supported schemes: `{PLAIN}` for plain text passwords, `{SHA}` for SHA-1 hashing, and `{SSHA}` for salted SHA-1 hashing.
- `password` : the password
- `comment` : a plain text comment for your own use

If you fail to specify a scheme, the password will need to be encrypted with the `crypt(3)` function, for example with the help of the `htpasswd` command-line utility from the Apache packages.

Note

If you aren't too keen on installing Apache on your system just for the sake of the `htpasswd` tool, you may resort to online tools, as there are plenty of them available. Fire up your favorite search engine and type `online htpasswd` .

Access

Two important directives are brought up by this module: `allow` and `deny` . They let you allow or deny access to a resource for a specific IP address or IP address range.

Both directives have the same syntax: `allow IP | CIDR | unix: | all` , where `IP` is an IP address, `CIDR` is an IP address range (CIDR syntax), unix: represents all UNIX domain sockets, and `all` specifies that the directive applies to all clients:

Copy

```
location {
    allow 127.0.0.1; # allow local IP address
    allow unix:; # allow UNIX domain sockets
    deny all; # deny all other IP addresses
}
```

Note that rules are processed from top-down—if your first instruction is `deny all` , all possible `allow` exceptions that you place afterwards will have no effect. The opposite is also true—if you start with `allow all` , all possible `deny` directives that you place afterwards will have no effect, as you already allowed all the IP addresses.

Limit connections

The mechanism induced by this module is a little more complex than the regular ones. It allows you to define the maximum number of simultaneous connections to the server for a specific **zone**.

The first step is to define the zone using the `limit_conn_zone` directive:

- Directive syntax: `limit_conn_zone $variable zone=name:size;`
- `$variable` is the variable that will be used to differentiate one client from another, typically `$binary_remote_addr` —the IP address of the client in the binary format (this is more efficient than ASCII)
- `name` is an arbitrary name given to the zone
- `size` is the maximum size you allocate to the table storing session states

The following example defines the zones based on the client IP addresses:

Copy

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

Now that you have defined a zone, you may limit the connections using `limit_conn` :

Copy

```
limit_conn zone_name connection_limit;
```

When applied to the previous example, it becomes:

Copy

```
location /downloads/ {
    limit_conn myzone 1;
}
```

As a result, requests that share the same `$binary_remote_addr` are subject to the connection limit (one simultaneous connection). If the limit is reached, all additional concurrent requests will be answered with a `503 Service unavailable` HTTP response. This response code can be overridden if you specify another code via the `limit_conn_status` directive. If you wish to log client requests that are affected by the limits you have set, enable the `limit_conn_log_level` directive, and specify the log level (`info | notice | warn | error`).

Limit request

In a similar fashion, the **Limit request** module allows you to limit the number of requests for a defined zone.

Defining the zone is done via the `limit_req_zone` directive; its syntax differs from the **Limit zone** equivalent directive:

Copy

```
limit_req_zone $variable zone=name:max_memory_size rate=rate;
```

The directive parameters are identical except for the trailing `rate` : expressed in requests per second (r/s) or requests per minute (r/m). It defines a request rate that will be applied to clients where the zone is enabled. To apply a zone to a location, use the `limit_req` directive:

Copy

```
limit_req zone=name burst=burst [nodelay];
```

The `burst` parameter defines the maximum possible bursts of requests—when the amount of requests received from a client exceeds the limit defined in the zone, the responses are delayed in a manner that respects the rate that you defined. To a certain extent, only a maximum of `burst` requests will be accepted simultaneously. Past this limit, Nginx returns a `503 Service Unavailable` HTTP error response. This response code can be overridden if you specify another code via the `limit_req_status` directive.

Copy

```
limit_req_zone $binary_remote_addr zone=myzone:10m rate=2r/s;
[…]
location /downloads/ {
  limit_req zone=myzone burst=10;
  limit_req_status 404; # returns a 403 error if limit is exceeded
}
```

If you wish to log client requests that are affected by the limits you have set, enable the `limit_req_log_level` directive, and specify the log level (`info | notice | warn | error`).

Auth_request

The **auth_request** module was implemented in the recent versions of Nginx, and allows you to allow or deny access to a resource based on the result of a sub-request. Nginx calls the URI that you specify via the `auth_request` directive: if the sub-request returns a 2XX response code (that is, `HTTP/200 OK`), access is allowed. If the sub-request returns a 401 or 403 status code, access is denied, and Nginx forwards the response code to the client. Should the backend return any other response code, Nginx will consider it to be an error and deny access to the resource.

Copy

```
location /downloads/ {
    # if the script below returns a 200 status code,
    # the download is authorized
    auth_request /authorization.php;
}
```

Additionally, the module offers a second directive called `auth_request_set` , allowing you to set a variable after the sub-request is executed. You can insert variables that originate from the sub-request upstream `($upstream_http_*`) such as `$upstream_http_server` or other HTTP headers from the server response.

Copy

```
location /downloads/ {
    # requests authorization from PHP script
    auth_request /authorization.php;
    # assuming authorization is granted, get filename from
    # sub-request response header and redirect
    auth_request_set $filename "${upstream_http_x_filename}.zip";
    rewrite ^ /documents/$filename;
}
```

Content and encoding

The following set of modules provides functionalities having an effect on the contents served to the client, either by modifying the way the response is encoded, by affecting the headers, or by generating a response from scratch.

Empty GIF

The purpose of this module is to provide a directive that serves a **1 x 1** transparent GIF image from the memory. Such files are sometimes used by web designers to tweak the appearance of their website. With this directive, you get an empty GIF straight from the memory instead of reading and processing an actual GIF file from the storage space.

To utilize this feature, simply insert the `empty_gif` directive in the location of your choice:

Copy

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

FLV and MP4

FLV and MP4 are separate modules enabling a simple functionality that becomes useful when serving Flash (FLV) or MP4 video files. It parses a special argument of the request, `start` , which indicates the offset of the section that the client wishes to download or pseudo-stream. The video file must thus be accessed with the following URI: `video.flv?start=XXX` . This parameter is prepared automatically by mainstream video players such as JWPlayer.

Note

This module is not included in the default Nginx build.

To utilize this feature, simply insert the `flv` or `mp4` directive in the location of your choice:

Copy

```
location ~* \.flv {
    flv;
}
location ~* \.mp4 {
    mp4;
}
```

Be aware that in case Nginx fails to seek the requested position within the video file, the request will result in a `500 Internal Server Error` HTTP response. JWPlayer sometimes misinterprets this error, and simply displays a **Video not found** error message.

HTTP headers

Two directives are introduced by this module that affect the header of the response sent to the client.

First, `add_header Name value [always]` lets you add a new line in the response headers, respecting the following syntax: `Name: value` . The line is added only for responses with the following codes: `200` , `201` , `204` , `301` , `302` , and `304` . You may insert variables in the `value` argument. If you specify `always` at the end of the directive value, the header will always be added regardless of the response code.

Additionally, the `expires` directive allows you to control the value of the **Expires and Cache-Control HTTP header** sent to the client, affecting the requests of the codes listed previously. It accepts a single value among the following:

- `off` : Does not modify either of the headers

- A time value: The expiration date of the file is set to **the current time** +, **the time you specify**. For example, `expires 24h` will return an expiry date set to 24 hours from now
- `epoch` : The expiration date of the file is set to January 1, 1970. The Cache-Control header is set to `no-cache`
- `max` : The expiration date of the file is set to December 31, 2037. The Cache-Control header is set to 10 years

Addition

The Addition module allows you (through simple directives) to add content before or after the body of the HTTP response.

Note

This module is not included in the default Nginx build.

The two main directives are:

Copy

```
add_before_body file_uri;
add_after_body file_uri;
```

As stated previously, Nginx triggers a sub-request for fetching the specified URI. Additionally, you can define the type of files to which the content is appended in case your `location` block pattern is not specific enough (default: `text/html`):

Copy

```
addition_types mime_type1 [mime_type2…];
addition_types *;
```

Substitution

Along the same lines as that of the preceding module, the Substitution module allows you to search and replace text directly from the response body:

Copy

```
sub_filter searched_text replacement_text;
```

Note

This module is not included in the default Nginx build.

Two additional directives provide more flexibility:

- `sub_filter_once` (`on` or `off` , default `on`): Only replaces the text once, and stops after the first occurrence.
- `sub_filter_types` (default `text/html`): Affects the additional MIME types that are eligible for text replacement. The `*` wildcard is allowed.

Gzip filter

This module allows you to compress the response body with the Gzip algorithm before sending it to the client. To enable Gzip compression, use the `gzip` directive (`on` or `off`) at the `http` , `server` , `location` , and even the `if` level (though that is not recommended). The following directives will help you further configure the filter options:

Directive	Description
`gzip_buffers` Context: `http` , `server` , `location`	Defines the number and size of buffers to be used for storing the compressed response. Syntax: `gzip_buffers amount size;` Default: `gzip_buffers 4 4k` (or `8k` depending on the OS).

Directive	Description
`gzip_comp_level` Context: `http` , `server` , `location`	Defines the compression level of the algorithm. The specified value ranges from 1 (low compression, faster for the CPU) to 9 (high compression, slower). Syntax: Numeric value. Default: `1`
`gzip_disable` Context: `http` , `server` , `location`	Disables Gzip compression for the requests where the User-Agent HTTP header matches the specified regular expression. Syntax: Regular expression Default: None
`gzip_http_version` Context: `http` , `server` , `location`	Enables Gzip compression for the specified protocol version. Syntax: `1.0` or `1.1` Default: `1.1`
`gzip_min_length` Context: `http` , `server` , `location`	If the response body length is inferior to the specified value, it is not compressed. Syntax: Numeric value (size) Default: `0`
`gzip_proxied` Context: `http` , `server` , `location`	Enables or disables Gzip compression for the body of responses received from a proxy (see reverse-proxying mechanisms in later chapters). The directive accepts the following parameters; some can be combined: • `off/any` : Disables or enables compression for all requests • `expired` : Enables compression if the **Expires** header prevents caching • `no-cache/no-store/private` : Enables compression if the **Cache-Control** header is set to no-cache, no-store, or private • `no_last_modified` : Enables compression in case the **Last-Modified** header is not set • `no_etag` : Enables compression in case the **ETag** header is not set • `auth` : Enables compression in case an **Authorization** header is set

`gzip_types`
Context: `http` , `server` , `location`
	Enables compression for types other than the default `text/html` MIME type.
	Syntax:
	Copy
	`gzip_types mime_type1 [mime_type2…];`
	`gzip_types *;`
	Default: `text/html` (cannot be disabled)

`gzip_vary` Context: `http` , `server` , `location`	Adds the **Vary: Accept-Encoding** HTTP header to the response. Syntax: `on` or `off` Default: `off`
`gzip_window` Context: `http` , `server` , `location`	Sets the size of the window buffer (`windowBits` argument) for Gzipping operations. This directive value is used for calls to functions from the Zlib library. Syntax: Numeric value (size) Default: `MAX_WBITS` constant from the Zlib library
`gzip_hash` Context: `http` , `server` , `location`	Sets the amount of memory that should be allocated for the internal compression state (`memLevel` argument). This directive value is used for calls to functions from the Zlib library. Syntax: Numeric value (size) Default: `MAX_MEM_LEVEL` constant from the Zlib prerequisite library

Directive	Description
`postpone_gzipping` Context: `http`, `server`, `location`	Defines a minimum data threshold to be reached before starting the Gzip compression. Syntax: Size (numeric value) Default: `0`
`gzip_no_buffer` Context: `http`, `server`, `location`	By default, Nginx waits until at least one buffer (defined by `gzip_buffers`) is filled with data before sending the response to the client. Enabling this directive disables buffering. Syntax: `on` or `off` Default: `off`

Gzip static

This module adds a simple functionality to the Gzip filter mechanism—when its `gzip_static` directive (`on`, `off`, or `always`) is enabled, Nginx will automatically look for a `.gz` file corresponding to the requested document before serving it. This allows Nginx to send pre-compressed documents instead of compressing documents on the fly at each request. Specifying `always` will force Nginx to serve the gzip version regardless of whether the client accepts gzip encoding.

Note

This module is not included in the default Nginx build.

If a client requests `/documents/page.html`, Nginx checks for the existence of a `/documents/page.html.gz` file. If the `.gz` file is found, it is served to the client. Note that Nginx does not generate `.gz` files itself, even after serving the requested files.

Gunzip filter

With the **Gunzip filter** module, you can decompress a gzip-compressed response sent from the backend in order to serve it **raw** to the client. For example, in cases where the client browser is not able to process the gzipped files (Microsoft Internet Explorer 6), simply insert `gunzip on;` in a location block to employ this module. You can also set the buffer amount and size with `gunzip_buffers amount size;` where `amount` is the amount of buffers to allocate, and `size` is the size of each allocated buffer.

Charset filter

With the **Charset filter** module, you can control the character set of the response body more accurately. Not only are you able to specify the value of the `charset` argument of the Content-Type HTTP header (such as `Content-Type: text/html; charset=utf-8`), but Nginx can also re-encode the data to a specified encoding method automatically.

Directive	Description
`charset` Context: `http`, `server`, `location`, `if`	This directive adds the specified encoding to the Content-Type header of the response. If the specified encoding differs from the `source_charset` one, Nginx re-encodes the document. Syntax: `charset encoding | off;` Default: `off` Example: `charset utf-8;`
`source_charset` Context: `http`, `server`, `location`, `if`	Defines the initial encoding of the response; if the value specified in the `charset` directive differs, Nginx re-encodes the document. Syntax: `source_charset encoding;`
`override_charset` Context: `http`, `server`, `location`, `if`	When Nginx receives a response from the proxy or FastCGI gateway, this directive defines whether or not the character encoding should be checked and potentially overridden. Syntax: `on` or `off` Default: `off`

Directive	Description

charset_types
Context: http , server , location
	Defines the MIME types that are eligible for re-encoding.
	Syntax:
	Copy
	`charset_types mime_type1 [mime_type2…];`
	`charset_types * ;`
	Default: text/html, text/xml, text/plain , text/vnd.wap.wml , application/x-javascript, application/rss+xml

charset_map Context: http	Lets you define character re-encoding tables. Each line of the table contains two hexadecimal codes to be exchanged. You will find re-encoding tables for the koi8-r character set in the default Nginx configuration folder (koi-win and koi-utf). Syntax: charset_map src_encoding dest_encoding { … }

Memcached

Memcached is a daemon application that can be connected to via sockets. Its main purpose, as the name suggests, is to provide an efficient distributed key/value memory caching system. The **Nginx Memcached** module provides directives allowing you to configure access to the Memcached daemon.

Directive	Description

memcached_pass
Context: location , if
	Defines the hostname and port of the Memcached daemon.
	Syntax: memcached_pass hostname:port;
	Example: memcached_pass localhost:11211;

memcached_bind
Context: http , server , location
	Forces Nginx to use the specified local IP address for connecting to the Memcached server. This can come in handy if your server has multiple network cards connected to different networks.
	Syntax: memcached_bind IP_address;
	Example: memcached_bind 192.168.1.2;

memcached_connect_timeout Context: http , server , location	Defines the connection timeout in milliseconds (default: 60,000). Example: memcached_connect_timeout 5000;
memcached_send_timeout Context: http , server , location	Defines the data writing operations timeout in milliseconds (default: 60,000). Example: memcached_send_timeout 5,000;
memcached_read_timeout Context: http , server , location	Defines the data reading operations timeout in milliseconds (default: 60,000). Example: memcached_read_timeout 5,000;
memcached_buffer_size Context: http , server , location	Defines the size of the read and write buffer in bytes (default: page size). Example: memcached_buffer_size 8k;
memcached_next_upstream Context: http , server , location	When the memcached_pass directive is connected to an upstream block (refer to the section on **upstream module**), this directive defines the conditions that should be matched in order to skip to the next upstream server. Syntax: Values selected among error timeout, invalid_response, not_found , or off Default: error timeout Example: memcached_next_upstream off;

Directive	Description
`memcached_gzip_flag` Context: `http` , `server` , `location`	Checks for the presence of the specified flag in the memcached server response. If the flag is present, Nginx sets the `Content-encoding` header to `gzip` to indicate that it will be serving gzipped content. Syntax: numeric flag Default: (none) Example: `memcached_gzip_flag 1;`

Additionally, you will need to define the `$memcached_key` variable, which defines the key of the element that you are placing or fetching from the cache. You may, for instance, use `set $memcached_key $uri` or `set $memcached_key $uri?$args` .

Note that the Nginx Memcached module is only able to retrieve data from the cache; it does not store the results of requests. Storing data in the cache should be done by a server-side script. You just need to make sure to employ the same key-naming scheme in both your server-side scripts and the Nginx configuration. As an example, we could decide to use `memcached` to retrieve data from the cache before passing the request to a proxy if the requested URI is not found (see Chapter 7, **From Apache to Nginx**, for more details about the Proxy module):

Copy

```
server {
    server_name example.com;
    […]
    location / {
        set $memcached_key $uri;
        memcached_pass 127.0.0.1:11211;
        error_page 404 @notcached;
    }
    location @notcached {
        internal;
        # if the file is not found, forward request to proxy
        proxy_pass 127.0.0.1:8080;
    }
}
```

Image filter

This module provides image processing functionalities through the **GD Graphics Library** (also known as **gdlib**).

Note

This module is not included in the default Nginx build.

Make sure to employ the following directives on a `location` block that filters image files only, such as `location ~* \.(png|jpg|gif)$ { … }` .

Directive	Description
`image_filter` Context: `location`	Lets you apply a transformation on the image before sending it to the client. There are five options available: • `test` : Makes sure that the requested document is an image file, returns a `415 Unsupported media type` HTTP error if the test fails. • `size` : Composes a simple JSON response indicating information about the image such as the size and type (for example, `{ "img": { "width":50, "height":50, "type":"png"}}`). If the file is invalid, a simple `{}` is returned. • `resize width height` : Resizes the image to the specified dimensions. • `crop width height` : Selects a portion of the image of the specified dimensions. • `rotate 90 | 180 | 270` : Rotates the image by the specified angle (in degrees). Example: `image_filter resize 200 100;`

Directive	Description
`image_filter_buffer` Context: `http` , `server` , `location`	Defines the maximum file size for the images to be processed. Default: `image_filter_buffer 1m;`
`image_filter_jpeg_quality` Context: `http` , `server` , `location`	Defines the quality of the output JPEG images. Default: `image_filter_jpeg_quality 75;`
`image_filter_transparency` Context: `http` , `server` , `location`	By default, PNG and GIF images keep their existing transparency during the operations that you perform by using the Image Filter module. If you set this directive to `off` , all existing transparency will be lost, but the image quality will be improved. Syntax: `on` or `off` Default: `on`
`image_filter_sharpen` Context: `http` , `server` , `location`	Sharpens the image by the specified percentage (value may exceed 100). Syntax: Numeric value Default: `0`
`image_filter_interlace` Context: `http` , `server` , `location`	Enables interlacing of the output image. If the output image is a JPG file, the image is generated in the **progressive JPEG** format. Syntax: `on` or `off` Default: `off`

Please note that when it comes to JPG images, Nginx automatically strips off the metadata (such as EXIF) if it occupies more than five percent of the total space of the file.

XSLT

The Nginx XSLT module allows you to apply an XSLT transform on an XML file or response received from a backend server (proxy, FastCGI, and so on) before serving the client.

Note

This module is not included in the default Nginx build

Directive	Description
`xml_entities` Context: `http` , `server` , `location`	Specifies the DTD file containing symbolic element definitions. Syntax: File path Example: `xml_entities xml/entities.dtd;`
`xslt_stylesheet` Context: `location`	Specifies the XSLT template file path with its parameters. Variables may be inserted in the parameters. Syntax: `xslt_stylesheet template [param1] [param2…];` Example: `xslt_stylesheet xml/sch.xslt param=value;`

Directive	Description

`xslt_types` Context: `http`, `server`, `location`	Defines the additional MIME types, other than `text/xml`, to which the transforms may apply. Syntax: MIME type Example:
	Copy
	`xslt_types text/xml text/plain;`
	`xslt_types *;`

`xslt_paramxslt_string_param` Context: `http`, `server`, `location`	Both the directives allow defining parameters for XSLT stylesheets. The difference lies in the way the specified value is interpreted: the XPath expressions in the value are processed using `xslt_param`, while `xslt_string_param` is used for plain character strings. Syntax: `xslt_param key value;`

About your visitors

The following set of modules provides extra functionality that helps you find out more information about the visitors by parsing client request headers for browser name and version, assigning an identifier to requests presenting similarities, and so on.

Browser

The Browser module parses the User-Agent HTTP header of the client request in order to establish values for the variables that can be employed later in the configuration. The three variables produced are:

- `$modern_browser` : If the client browser is identified as being a modern web browser, the variable takes the value defined by the `modern_browser_value` directive.
- `$ancient_browser` : If the client browser is identified as being an old web browser, the variable takes the value defined by `ancient_browser_value` .
- `$msie` : This variable is set to `1` if the client is using a Microsoft IE browser.

To help Nginx recognize the web browsers and for telling the old from the modern, you need to insert multiple occurrences of the `ancient_browser` and `modern_browser` directives:

Copy

```
modern_browser opera 10.0;
```

With this example, if the User-Agent HTTP header contains Opera 10.0, the client browser is considered modern.

Map

Just like the Browser module, the Map module allows you to create maps of values depending on a variable:

Copy

```
map $uri $variable {
  /page.html  0;
  /contact.html  1;
  /index.html  2;
  default 0;
}
rewrite ^ /index.php?page=$variable;
```

Note that the `map` directive can only be inserted within the `http` block. Following this example, `$variable` may have three different values. If `$uri` was set to `/page.html` , `$variable` is now defined as `0` ; if `$uri` was set to `/contact.html` , `$variable` is now `1` ; if `$uri` was set to `/index.html` , `$variable` now equals `2` . For all other cases (`default`), `$variable` is set to `0` . The last instruction rewrites the URL accordingly. Apart from `default` , the `map` directive accepts another special keyword: `hostnames` . It allows you to match the hostnames using wildcards such as `*.domain.com` .

Two additional directives allow you to tweak the way Nginx manages the mechanism in memory:

- `map_hash_max_size` : Sets the maximum size of the hash table holding a map
- `map_hash_bucket_size` : Sets the maximum size of an entry in the map

Regular expressions may also be used in patterns if you prefix them with `~` (case sensitive) or `~*` (case insensitive):

Copy

```
map $http_referer $ref {
    ~google "Google";
    ~* yahoo "Yahoo";
    \~bing "Bing"; # not a regular expression due to the \ before the tilde
    default $http_referer; # variables may be used
    }
```

Geo

The purpose of this module is to provide a functionality that is quite similar to the `map` directive—affecting a variable based on the client data (in this case, the IP address). The syntax is slightly different in that you are allowed to specify IPv4 and IPv6 address ranges (in CIDR format):

Copy

```
geo $variable {
  default unknown;
  127.0.0.1    local;
  123.12.3.0/24  uk;
  92.43.0.0/16  fr;
}
```

Note that the preceding block is being presented to you just for the sake of the example and does not actually detect U.K. and French visitors; you'll have to use the GeoIP module if you wish to achieve proper geographical location detection. In this block, you may insert a number of directives that are specific to this module:

- `delete` : Allows you to remove the specified subnetwork from the mapping.
- `default` : The default value given to `$variable` in case the user's IP address does not match any of the specified IP ranges.
- `include` : Allows you to include an external file.
- `proxy` : Defines a subnet of trusted addresses. If the user IP address is among the trusted ones, the value of the `X-Forwarded-For` header is used as an IP address instead of the socket IP address.
- `proxy_recursive` : If enabled, this will look for the value of the `X-Forwarded-For` header even if the client IP address is not trusted.
- `ranges` : If you insert this directive as the first line of your `geo` block, it allows you to specify IP ranges instead of CIDR masks. The following syntax is thus permitted: `127.0.0.1-127.0.0.255 LOCAL;`

GeoIP

Although the name suggests some similarities with the previous one, this optional module provides accurate geographical information about your visitors by making use of the **MaxMind** (http://www.maxmind.com (http://www.maxmind.com)) GeoIP binary databases. You need to download the database files from the MaxMind website and place them in your Nginx directory.

Note

This module is not included in the default Nginx build.

All you have to do then is specify the database path with one of the following directives:

Copy

```
geoip_country country.dat; # country information db
geoip_city city.dat; # city information db
geoip_org geoiporg.dat; # ISP/organization db
```

The first directive enables several variables: `$geoip_country_code` (two-letter country code), `$geoip_country_code3` (three-letter country code), and `$geoip_country_name` (full country name). The second directive includes the same variables, but provides additional information: `$geoip_region` , `$geoip_city` , `$geoip_postal_code` , `$geoip_city_continent_code` , `$geoip_latitude` , `$geoip_longitude` , `$geoip_dma_code` , `$geoip_area_code` , and `$geoip_region_name` . The third directive offers information about the organization or ISP that owns the specified IP address by filling up the `$geoip_org` variable.

Note

If you need the variables to be encoded in UTF-8, simply add the `utf8` keyword at the end of the `geoip_` directives.

UserID filter

This module assigns an identifier to the clients by issuing cookies. The identifier can be accessed from the variables `$uid_got` and `$uid_set` further in the configuration.

Directive	Description
`userid` Context: `http` , `server` , `location`	Enables or disables issuing and logging of cookies. The directive accepts four possible values: • `on` : Enables `v2` cookies and logs them • `v1` : Enables `v1` cookies and logs them • `log` : Does not send cookie data, but logs the incoming cookies • `off` : Does not send cookie data Default value: `userid off;`
`userid_service` Context: `http` , `server` , `location`	Defines the IP address of the server issuing the cookie. Syntax: `userid_service ip;` Default: IP address of the server
`userid_name` Context: `http` , `server` , `location`	Defines the name assigned to the cookie. Syntax: `userid_name name;` Default value: The user identifier
`userid_domain` Context: `http` , `server` , `location`	Defines the domain assigned to the cookie. Syntax: `userid_domain domain;` Default value: None (the domain part is not sent)
`userid_path` Context: `http` , `server` , `location`	Defines the path part of the cookie. Syntax: `userid_path path;` Default value: `/`
`userid_expires` Context: `http` , `server` , `location`	Defines the cookie expiration date. Syntax: `userid_expires date | max;` Default value: No expiration date
`userid_p3p` Context: `http` , `server` , `location`	Assigns a value to the P3P header sent with the cookie. Syntax: `userid_p3p data;` Default value: None

Referer

A simple directive is introduced by this module: `valid_referers`. Its purpose is to check the Referer HTTP header from the client request, and possibly, to deny access based on the value. If the referer is considered invalid, `$invalid_referer` is set to `1`. In the list of valid referers, you may employ three kinds of values:

- None: The absence of a referer is considered to be a valid referer
- Blocked: A masked referer (such as `XXXXX`) is also considered valid
- A server name: The specified server name is considered to be a valid referer

Following the definition of the `$invalid_referer` variable, you may, for example, return an error code if the referer was found invalid:

Copy

```
valid_referers none blocked *.website.com *.google.com;
if ($invalid_referer) {
  return 403;
}
```

Be aware that spoofing the Referer HTTP header is a very simple process, so checking the referer of client requests should not be used as a security measure.

Two more directives are offered by this module: `referer_hash_bucket_size` and `referer_hash_max_size`, which allow you to define the bucket size and maximum size of the valid referers hash tables respectively.

Real IP

This module provides one simple feature—it replaces the client IP address by the one specified in the **X-Real-IP** HTTP header for clients that visit your website behind a proxy, or for retrieving IP addresses from the proper header if Nginx is used as a backend server (it essentially has the same effect as Apache's `mod_rpaf`; see Chapter 7, **From Apache to Nginx**, for more details). To enable this feature, you need to insert the `real_ip_header` directive that defines the HTTP header to be exploited—either `X-Real-IP` or `X-Forwarded-For`. The second step is to define the trusted IP addresses, in other words, the clients that are allowed to make use of those headers. This can be done thanks to the `set_real_ip_from` directive, which accepts both IP addresses and CIDR address ranges:

Copy

```
real_ip_header X-Forwarded-For;
set_real_ip_from 192.168.0.0/16;
set_real_ip_from 127.0.0.1;
set_real_ip_from unix:; # trusts all UNIX-domain sockets
```

Note

This module is not included in the default Nginx build.

Split Clients

The Split Clients module provides a resource-efficient way to split the visitor base into subgroups based on the percentages that you specify. To distribute the visitors into one group or another, Nginx hashes a value that you provide (such as the visitor's IP address, cookie data, query arguments, and so on), and decides which group the visitor should be affected to. The following example configuration divides the visitors into three groups based on their IP address. If a visitor is affected to the first 50 percent, the value of `$variable` will be set to `group1`:

Copy

```
split_clients "$remote_addr" $variable {
  50% "group1";
  30% "group2";
  20% "group3";
}
location ~ \.php$ {
  set $args "${query_string}&group=${variable}";
}
```

SSL and security

Nginx provides secure HTTP functionalities through the SSL module, but also offers an extra module called **Secure Link** that helps you protect your website and visitors in a totally different way.

SSL

The SSL module enables HTTPS support, HTTP over SSL/TLS in particular. It gives you the option to serve secure websites by providing a certificate, a certificate key, and other parameters defined with the following directives:

Note

This module is not included in the default Nginx build.

Directive	Description
`ssl` Context: `http` , `server`	Enables HTTPS for the specified server. This directive is the equivalent of `listen 443 ssl` or `listen port ssl` more generally. Syntax: `on` or `off` Default: `ssl off;`
`ssl_certificate` Context: `http` , `server`	Sets the path of the PEM certificate. Syntax: File path
`ssl_certificate_key` Context: `http` , `server`	Sets the path of the PEM secret key file. Syntax: File path
`ssl_client_certificate` Context: `http` , `server`	Sets the path of the client PEM certificate. Syntax: File path
`ssl_crl` Context: `http` , `server`	Orders Nginx to load a CRL (Certificate Revocation List) file, which allows checking the revocation status of certificates.
`ssl_dhparam` Context: `http` , `server`	Sets the path of the **Diffie-Hellman** parameters file. Syntax: File path.
`ssl_protocols` Context: `http` , `server`	Specifies the protocol that should be employed. Syntax: `ssl_protocols [SSLv2] [SSLv3] [TLSv1] [TLSv1.1] [TLSv1.2];` Default: `ssl_protocols TLSv1 TLSv1.1 TLSv1.2;`
`ssl_ciphers` Context: `http` , `server`	Specifies the ciphers that should be employed. The list of available ciphers can be obtained by running the following command from the shell: `openssl ciphers` . Syntax: `ssl_ciphers cipher1[:cipher2…];` Default: `ssl_ciphers ALL:!ADH:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP;`

Directive	Description
`ssl_prefer_server_ciphers` Context: `http` , `server`	Specifies whether server ciphers should be preferred over client ciphers. Syntax: `on` or `off` Default: `off`
`ssl_verify_client` Context: `http` , `server`	Enables verifying certificates to be transmitted by the client and sets the result in the `$ssl_client_verify` . The `optional_no_ca` value verifies the certificate if there is one, but does not require it to be signed by a trusted CA certificate. Syntax: `on` | `off` | `optional` | `optional_no_ca` Default: `off`
`ssl_verify_depth` Context: `http` , `server`	Specifies the verification depth of the client certificate chain. Syntax: Numeric value Default: `1`
`ssl_session_cache` Context: `http` , `server`	Configures the cache for SSL sessions. Syntax: `off` , `none` , `builtin:size` or `shared:name:size` Default: `off` (disables SSL sessions)
`ssl_session_timeout` Context: `http` , `server`	When the SSL sessions are enabled, this directive defines the timeout for using session data. Syntax: Time value Default: 5 minutes
`ssl_password_phrase` Context: `http` , `server`	Specifies a file containing the passphrases for secret keys. Each passphrase is specified on a separate line; they are tried one after the other when loading a certificate key. Syntax: file name Default: (none)
`ssl_buffer_size` Context: `http` , `server`	Specifies the buffer size when serving requests over SSL. Syntax: Size value Default: 16k
`ssl_session_tickets` Context: `http` , `server`	Enables TLS session tickets, allowing the client to reconnect faster by skipping re-negotiation. Syntax: `on` or `off` Default: `on`
`ssl_session_ticket_key` Context: `http` , `server`	Sets the path of the key file used to encrypt and decrypt the TLS session tickets. By default, a random value is generated. Syntax: file name Default: (none)

Directive	Description
`ssl_trusted_certificate` Context: `http` , `server`	Sets the path of a trusted certificate file (PEM format) used to validate the authenticity of client certificates as well as the stapling of OCSP responses. More about SSL stapling can be found later on in the chapter. Syntax: file name Default: (none)

Additionally, the following variables are made available:

- `$ssl_cipher` : Indicates the cipher used for the current request
- `$ssl_client_serial` : Indicates the serial number of the client certificate
- `$ssl_client_s_dn` and `$ssl_client_i_dn` : Indicates the value of the Subject and Issuer DN of the client certificate
- `$ssl_protocol` : Indicates the protocol in use for the current request
- `$ssl_client_cert` and `$ssl_client_raw_cert` : Returns the client certificate data, which is raw data for the second variable
- `$ssl_client_verify` : Set to `SUCCESS` if the client certificate was successfully verified
- `$ssl_session_id` : Allows you to retrieve the ID of an SSL session

Setting up an SSL certificate

Although the SSL module offers a lot of possibilities, in most cases only a couple of directives are actually useful for setting up a secure website. This guide will help you to configure Nginx to use an SSL certificate for your website (in the example, your website is identified by `secure.website.com`). Before doing so, ensure that you already have the following elements at your disposal:

- A `.key` file generated with the following command: `openssl genrsa -out secure.website.com.key 1024` (other encryption levels work too).
- A `.csr` file generated with the following command: `openssl req -new -key secure.website.com.key -out secure.website.com.csr` .
- Your website certificate file, as issued by the Certificate Authority, for example, `secure.website.com.crt` . (Note: In order to obtain a certificate from the CA, you will need to provide your `.csr` file.)
- The CA certificate file as issued by the CA (for example, `gd_bundle.crt` if you purchased your certificate from http://www.GoDaddy.com (http://www.GoDaddy.com)).

The first step is to merge your website certificate and the CA certificate together with the following command:

Copy

```
cat secure.website.com.crt gd_bundle.crt > combined.crt
```

You are then ready to configure Nginx for serving secure content:

Copy

```
server {
    listen 443;
    server_name secure.website.com;
    ssl on;
    ssl_certificate /path/to/combined.crt;
    ssl_certificate_key /path/to/secure.website.com.key;
    […]
}
```

SSL Stapling

SSL Stapling, also called **Online Certificate Status Protocol** (**OCSP**) Stapling, is a technique that allows clients to easily connect and resume sessions to an SSL/TLS server without having to contact the Certificate Authority, thus reducing the SSL negotiation time. In normal OCSP transactions, the client normally contacts the Certificate Authority so as to check the revocation status of the server's certificate. In the case of high traffic websites, this can cause a huge stress on the CA servers. An intermediary solution was designed—Stapling. The OCSP record is obtained periodically from the CA by your server itself, and is **stapled** to exchanges with the client. The OCSP record is cached by your server for a period of up to 48 hours in order to limit communications with the CA.

Enabling SSL Stapling should thus speed up the communication between your visitors and your server. Achieving this in Nginx is relatively simple: all you really need is to insert three directives in your server block, and obtain a full trusted certificate chain file (containing both the root and intermediate certificates) from your CA.

- `ssl_stapling on` : enables SSL Stapling within the server block
- `ssl_stapling_verify on` : enables verification of OCSP responses by the server
- `ssl_trusted_certificate filename` : where `filename` is the path of your full trusted certificate file (extension should be .pem).

Two optional directives also exist, which allow you to modify the behavior of this module:

- `ssl_stapling_file filename` : where `filename` is the path of a cached OCSP record, overriding the record provided by the OCSP responder specified in the certificate file.
- `ssl_stapling_responder url` : where `url` is the URL of your CA's OCSP responder, overriding the URL specified in the certificate file.

If you are having issues connecting to the OCSP responder, make sure your Nginx configuration contains a valid DNS resolver (using the `resolver` directive).

SPDY

The SPDY module offers support for the SPDY protocol (the SPDY module is not included by default). You can enable SPDY on your server by appending the keyword `spdy` at the end of your `listen` directive.

Copy

```
server {
    listen 443 ssl spdy;
    […]
}
```

Due to the nature of SPDY, it can only be enabled over SSL. Two directives and two variables are brought in by this module:

- `spdy_chunk_size` : sets the size of the SPDY chunks
- `spdy_headers_comp` : sets the compression level for headers (0 to disable, 1 to 9 from lowest/fastest to highest/slowest compression)
- `$spdy` : this variable contains the SPDY protocol version if SPDY is used, an empty string otherwise
- `$spdy_request_priority` : this variable indicates the request priority if SPDY is used, an empty string otherwise

Note

SPDY is a protocol developed by Google, aiming to improve web latency and security. Although its utility was demonstrated (albeit not always significantly), Google decided to abandon the project after the HTTP/2 standard was ratified. As a result, SPDY support will be officially withdrawn during the first half of 2016.

Secure link

Totally independent from the SSL module, Secure link provides basic protection by checking the presence of a specific hash in the URL before allowing the user to access a resource:

Copy

```
location /downloads/ {
secure_link_md5 "secret";
secure_link $arg_hash,$arg_expires;
    if ($secure_link = "") {
      return 403;
    }
}
```

With such a configuration, documents in the `/downloads/` folder must be accessed via a URL containing a query string parameter `hash=XXX` (note the `$arg_hash` in the example), where `XXX` is the MD5 hash of the secret you defined through the `secure_link_md5` directive. The second argument of the `secure_link` directive is a UNIX timestamp defining the expiration date. The `$secure_link` variable is empty if the URI does not contain the proper hash or if the date has expired. Otherwise, it is set to 1.

Note

This module is not included in the default Nginx build.

Other miscellaneous modules

The remaining three modules are optional (all three need to be enabled at compile time), and provide additional advanced functionality.

Stub status

The Stub status module was designed to provide information about the current state of the server, such as the amount of active connections, the total handled requests, and more. To activate it, place the `stub_status` directive in a `location` block. All requests matching the `location` block will produce the status page:

Copy

```
location = /nginx_status {
    stub_status on;
    allow 127.0.0.1; # you may want to protect the information
    deny all;
}
```

Note

This module is not included in the default Nginx build.

An example result produced by Nginx:

Copy

```
Active connections: 1
server accepts handled requests
 10 10 23
Reading: 0 Writing: 1 Waiting: 0
```

It's interesting to note that there are several server monitoring solutions, such as **Monitorix**, that offer Nginx support through the Stub status page by calling it at regular intervals and parsing the statistics.

Degradation

The HTTP Degradation module configures your server to return an error page when your server runs low on memory. It works by defining a memory amount that is to be considered low, and then specifies the locations for which you wish to enable the degradation check:

Copy

```
degradation sbrk=500m; # to be inserted at the http block level
degrade 204; # in a location block, specify the error code (204 or 444) to return in case the server condition has degraded
```

Google-perftools

This module interfaces the Google Performance Tools profiling mechanism for the Nginx worker processes. The tool generates a report based on the performance analysis of the executable code. More information can be discovered from the official website of the project http://code.google.com/p/google-perftools/ (http://code.google.com/p/google-perftools/).

Note

This module is not included in the default Nginx build.

In order to enable this feature, you need to specify the path of the report file that will be generated using the `google_perftools_profiles` directive:

Copy

```
google_perftools_profiles logs/profiles;
```

WebDAV

WebDAV is an extension of the well-known HTTP protocol. While HTTP was designed for visitors to download resources from a website (in other words, reading data), WebDAV extends the functionality of web servers by adding write operations such as creating files and folders, moving and copying files, and more. The Nginx WebDAV module implements a small subset of the WebDAV protocol:

Note

This module is not included in the default Nginx build.

Directive	Description
`dav_methods` Context: `http` , `server` , `location`	Selects the DAV methods you want to enable. Syntax: `dav_methods [off | [PUT] [DELETE] [MKCOL] [COPY] [MOVE]];` Default: `off`
`dav_access` Context: `http` , `server` , `location`	Defines access permissions at the current level. Syntax: `dav_access [user:r|w|rw] [group:r|w|rw] [all:r|w|rw];` Default: `dav_access user:rw;`
`create_full_put_path` Context: `http` , `server` , `location`	This directive defines the behavior when a client requests creation of a file in a directory that does not exist. If set to `on` , the directory path is created. If set to `off` , the file creation fails. Syntax: `on` or `off` Default: `off`
`min_delete_depth` Context: `http` , `server` , `location`	This directive defines a minimum URI depth for deleting files or directories when processing the `DELETE` command. Syntax: Numeric value Default: `0`

Third-party modules

The Nginx community has been growing larger over the past few years, and many additional modules have been written by third-party developers. These can be downloaded from the official wiki website http://wiki.nginx.org/nginx3rdPartyModules (http://wiki.nginx.org/nginx3rdPartyModules).

The currently available modules offer a wide range of new possibilities, among which are the following:

- An **Access Key** module to protect your documents in a fashion similar to Secure link, by **Mykola Grechukh**
- A **Fancy Indexes** module that improves the automatic directory listings generated by Nginx, by **Adrian Perez de Castro**
- The **Headers More** module, which improves flexibility with HTTP headers, by **Yichun Zhang** (**agentzh**)

- Many more features for various parts of the web server

To integrate a third-party module into your Nginx build, you need to follow these three simple steps:

1. Download the `.tar.gz` archive associated with the module that you wish to download.
2. Extract the archive with the following command: `tar xzf module.tar.gz` .
3. Configure your Nginx build with the following command:

   Copy

   ```
   ./configure --add-module=/module/source/path […]
   ```

Once you have finished building and installing the application, the module is available just like a regular Nginx module with its directives and variables.

If you are interested in writing Nginx modules yourself, **Evan Miller** published an excellent walkthrough: **Emiller's Guide to Nginx Module Development**. The complete guide may be consulted from his personal website at http://www.evanmiller.org/ (http://www.evanmiller.org/).

Next Section (/mapt/book/networking-and-servers/9781785280337/4/ch04lvl1sec27/summary)