Nginx Configuration Docs

Nginx does not have an equivalent to `.htaccess` configuration files to override default configurations. Every change that you want to make must go into the main server configuration, either a server or a location block.

See the Nginx documentation for more information about configuration instructions, and Understanding the Nginx Configuration File Structure and Configuration Contexts for more inforation about the different contexts where you can put your configuration.

Checking configuration and restaryting Nginx

Because Nginx uses a single configuration file we want to test the configuration before pushing it into production. To test the configuration, run the following command:

```
nginx -T
```

You may need to run the command with sudo, like this:

```
sudo nginx -T
```

If the tests are successful then you can restart the server to enable the configuration changes.

```
sudo systemctl restart nginx #systemd
sudo service nginx restart   #sysv init
```

Redirects

Nginx provides three different ways to do redirections: `retur`, `rewrite` and `try_files`. They each do different things with different levels of difficulty.

This guide will only cover `return`. For the other directives see Creating NGINX Rewrite Rules.

return

`return` is the first and simplest way to do redirects. For basic 3xx series redirects, it takes the numerical code for the redirect and the new URL to redirect to.

For a code in the 3xx series, the url parameter defines the new (rewritten) URL.

```
return (301 | 302 | 303 | 307) url;
```

For other codes, you optionally define a text string which appears in the body of the response (the standard text for the HTTP code, such as Not Found for 404, is still included in the header). The text can contain NGINX variables.

```
return (1xx | 2xx | 4xx | 5xx) ["text"];
```

This example that redirects clients to a new domain name.

```
server {
  listen 80;
  listen 443 ssl;
  server_name www.old-example.com;
  return 301 $scheme://www.new-name.com$request_uri;
}
```

The rewritten URL uses two NGINX variables to capture and replicate values from the original request URL: `$scheme` is the protocol (http or https) and `$request_uri` is the full URI including arguments.

To convert Apache rewrites to their Nginx equivalens see Converting Apache Rewrite Rules to NGINX Rewrite Rules

Redirect all traffic to HTTPS

If you haven't done so, you should redirect all traffic to HTTPS to increase the site's security and to allow access to modern APIs like service workers.

The following server snippet:

1. Defines the to listening ports, 80 for regular traffic and 443 for SSL
2. Sets the server name
   Uses a 301 (moved permanently) redirect to point to the HTTPS version of the resource

This snippet assumes that HTTPS is properly configured for your server. See Configuring HTTPS Servers in the Nginx documentation for instructionws on how to configure HTTPS for your server.

```
server {
  listen 80 default_server;
  listen 443 ssl;
  server_name example.com;
  return 301 https://$host$request_uri;
}
```

Redirecting from www. URLs

There are times when we want to move content from `www.example.com` to `example.com`. This is one way of doing it.

Instead of using the `$host` variable we specify the name of the host we're redirecting to.

```
server {
  server_name www.example.com;
  return 301 $scheme://example.com$request_uri;
}
```

You should not have the same content accessible from different URLs. This will cause problems with search engines and SEO. If you must keep the same content on both side, then use Canonical URLs to tell search engines which URL they should show users in the search results.

Inserting the www. at the beginning of URLs

There may be situations where we want to redirect all traffic to the `www.example.com` URL. For this we specify the name of the URL.

```
server {
  server_name example.com;
  return 301 $scheme://www.example.com$request_uri;
}
```

Remember that you shouldn't duplicate content in different sites under the same domain. If you must keep the content at different locations, then use Canonical URLs.

Cross-origin resources

The first set of directives control CORS (Cross-Origin Resource Sharing) access to resources from the server. CORS is an HTTP-header based mechanism that allows a server to indicate the external origins (domain, protocol, or port) which a browser should permit loading of resources.

For security reasons, browsers restrict cross-origin HTTP requests initiated from scripts. For example, XMLHttpRequest and the Fetch API follow the same-origin policy. A web application using those APIs can only request resources from the same origin the application was loaded from unless the response from other origins includes the apprpriate CORS headers.

General CORS access

The following directive will allow external clients to access all resources from your server.

```
add_header Access-Control-Allow-Origin '*'
```

Think very carefully if this is what you want to do as this will give everyone permission to embed your content on their sites. It may work better if you limit the permission to hosts you define, using something like this code:

The code will first check if the domain is one of the authorized domains. If it is, it will set the variable `$cors` to true.

The second block will only add the headers if `$cors` is true, meaning the request is coming from an authorized host.

```
set $cors '';
if ($http_origin ~
'^https?://(localhost|www\.yourdomain\.com|www\.yourotherdoma
in\.com)') {
  set $cors 'true';
}

if ($cors = 'true') {
  add_header 'Access-Control-Allow-Origin' "$http_origin"
always;
  add_header 'Access-Control-Allow-Credentials' 'true'
always;
  add_header 'Access-Control-Allow-Methods' 'GET, POST, PUT,
DELETE, OPTIONS' always;
  add_header 'Access-Control-Allow-Headers'
'Accept,Authorization,Cache-Control,Content-Type,DNT,If-
Modified-Since,Keep-Alive,Origin,User-Agent,X-Requested-
With' always;
  # required to be able to read Authorization header in
frontend
  #add_header 'Access-Control-Expose-Headers'
'Authorization' always;
}

if ($request_method = 'OPTIONS') {
  # Tell client that this pre-flight info is valid for 20
days
  add_header 'Access-Control-Max-Age' 1728000;
  add_header 'Content-Type' 'text/plain charset=UTF-8';
  add_header 'Content-Length' 0;
  return 204;
}
```

The code to conditionally add CORS headers is taken from CORS header support.

Cross-origin resource timing

The Resource Timing Level 1 specification defines an interface for web applications to access the complete timing information for resources in a document.

The Timing-Allow-Origin response header specifies origins that are allowed to see values of attributes retrieved via features of the Resource Timing API, which

would otherwise be reported as zero due to cross-origin restrictions.

If a resource isn't served with a `Timing-Allow-Origin` or if the header does not include the origin making the request some of the attributes of the `PerformanceResourceTiming` object will be set to zero.

```
add_header Timing-Allow-Origin "*";
```

Cross-origin images

As reported in the Chromium Blog and documented in Allowing cross-origin use of images and canvas can lead to fingerprinting attacks.

To mitigate the possibility of these attacks, you should use the `crossorigin` attribute in the images you request and the code snippet below to set the CORS header from the server.

```
location ~*
\.(bmp|cur|gif|ico|jpg|jpeg|png|apng|svg|webp|heic|heif|avif)
$ {
  add_header Access-Control-Allow-Origin '*';
}
```

Cross-origin web fonts

Google Chrome's Google Fonts troubleshooting guide tells us that, while Google Fonts may send the CORS header with every response, some proxy servers may strip it before the browser can use it to render the font.

```
location ~* \.(eot|otf|ttf|woff|woff2)$ {
    add_header
  Access-Control-Allow-Origin '*';
}
```

Custom Error Pages/Messsages

The following snippets use the `error_page` directive to point to the location of

custom error pages.

Both examples use `internal` to specify that the matching directive will only handle internal requests. You cannot call `errors/404.html` or `errors/50x.html` directly.

```
server {
  error_page 404 /errors/404.html;
  location = /errors/404.html {
    internal;
  }

  error_page 500 502 503 504 /errors/50x.html;
  location = /errors/50x.html {
    internal;
  }
}
```

Media Types and Character Encodings

Serve resources with the proper media types (a.k.a MIME types)

```
include mime.types;
default_type application/octet-stream;
```

```
types {

# Data interchange

  application/atom+xml         atom;
  application/json             json map topojson;
  application/ld+json          jsonld;
  application/rss+xml          rss;
  # Normalize to standard type.
```

```
  application/geo+json         geojson;
  application/xml              xml;
  # Normalize to standard type.
  application/rdf+xml          rdf;
# JavaScript
  # Servers should use text/javascript for JavaScript
resources.
  text/javascript              js mjs;
  application/wasm             wasm;
# Manifest files
  application/manifest+json    webmanifest;
  application/x-web-app-manifest+json   webapp;
  text/cache-manifest          appcache;

# Media files
  audio/midi                   mid midi kar;
  audio/mp4                    aac f4a f4b m4a;
  audio/mpeg                   mp3;
  audio/ogg                    oga ogg opus;
  audio/x-realaudio            ra;
  audio/x-wav                  wav;
  audio/x-matroska             mka;
  image/bmp                    bmp;
  image/gif                    gif;
  image/jpeg                   jpeg jpg;
  image/jxr                    jxr hdp wdp;
  image/png                    png;
  image/svg+xml                svg svgz;
  image/tiff                   tif tiff;
  image/vnd.wap.wbmp           wbmp;
  image/webp                   webp;
  image/x-jng                  jng;
  video/3gpp                   3gp 3gpp;
  video/mp4                    f4p f4v m4v mp4;
  video/mpeg                   mpeg mpg;
  video/ogg                    ogv;
  video/quicktime              mov;
  video/webm                   webm;
  video/x-flv                  flv;
  video/x-mng                  mng;
  video/x-ms-asf               asf asx;
  video/x-ms-wmv               wmv;
  video/x-msvideo              avi;
```

```
  video/x-matroska              mkv mk3d;

# Web fonts
  font/woff                       woff;
  font/woff2                      woff2;
  application/vnd.ms-fontobject   eot;
  font/ttf                        ttf;
  font/collection                 ttc;
  font/otf                        otf;
```

Set the default Charset attribute

Unless you have a good reason, and know what you're doing, you should set the character set of all text files to UTF-8 using the charset and source_charset directives.

charset is the initial character set for the document

```
charset utf-8;
source_charset utf-8
```

Frame Options

The example below sends the `X-Frame-Options` response header with the value DENY, informing browsers not to display the content of the web page in any frame to protect website against clickjacking.

While you could send the `X-Frame-Options` header for all of your website's pages, this has the potential downside that it forbids even any framing of your content (e.g.: when users visit your website using a Google Image Search results page).

Nonetheless, you should ensure that you send the `X-Frame-Options` header for all pages that allow a user to make a state-changing operation (e.g: pages that contain one-click purchase links, checkout or bank-transfer confirmation pages, pages that make permanent configuration changes, etc.).

```
# This sets the default value for the x_frame_options
variable on HTML elements
map $sent_http_content_type $x_frame_options {
  ~*text/html DENY;
}

# This uses the variable defined above
add_header X-Frame-Options $x_frame_options always;
```

Content Security Policy (CSP)

CSP (Content Security Policy) mitigates the risk of cross-site scripting and other content-injection attacks by setting a `Content Security Policy` which whitelists trusted sources of content for your website.

There is no policy that fits all websites, the example below is meant as a guideline for you to modify for your site.

The example policy below:

1. Restricts all fetches by default to the origin of the current website by setting the `default-src` directive to `'self'` - which acts as a fallback to all Fetch directives.
    1. This is convenient as you do not have to specify all Fetch directives that apply to your site, for example: `connect-src 'self'; font-src 'self'; script-src 'self'; style-src 'self'`, etc
    2. This restriction also means that you must explicitly define from which site(s) your website is allowed to load resources from, otherwise it will be restricted to the same origin as the page making the request
2. Disallows the <base> element on the website. This is to prevent attackers from changing the locations of resources loaded from relative URLs
    1. If you want to use the <base> element, then use `base-uri 'self'` instead
2. Only allows form submissions are from the current origin with: `form-action 'self'`
3. Prevents all websites (including your own) from embedding your webpages within e.g. the <iframe> or <object> element by setting: `frame-ancestors 'none'`.

    1. The `frame-ancestors`directive helps avoid "Clickjacking" attacks and is similar to the `X-Frame-Options` header
    2. Browsers that support the CSP header will ignore `X-Frame-Options` if `frame-ancestors` is also specified
4. Forces the browser to treat all the resources that are served over HTTP as if they were loaded securely over HTTPS by setting the `upgrade-insecure-requests` directive
    1. `upgrade-insecure-requests` **does not ensure HTTPS for the top-level navigation. If you want to force the website itself to be loaded over HTTPS you must include the** `Strict-Transport-Security` **header**
2. Includes the `Content-Security-Policy` header in all responses that are able to execute scripting. This includes the commonly used file types: HTML, XML and PDF documents. Although Javascript files can not execute scripts in a "browsing context", they are included to target web workers

   To make your CSP implementation easier, you can use an online CSP header generator. You should also use a validator to make sure your header does what you want it to do.

```
map $sent_http_content_type $content_security_policy {
    ~*text/(html|javascript)|application/pdf|xml
"default-src 'self'; base-uri 'none'; form-action
'self'; frame-ancestors 'none'; upgrade-insecure-
requests";
  }

add_header Content-Security-Policy
$content_security_policy always;
```

Directory access

Block access to hidden files and directories

Block access to files with sensitive

information

HTTP Strict Transport Security (HSTS)

Prevent some browsers from MIME-sniffing the response

Referrer Policy

Disable TRACE HTTP Method

Remove the X-Powered-By response header

Remove Apache-generated Server Information Footer

Fix broken AcceptEncoding Headers

Compress media types

Map extensions to media types

Cache expiration