Pitfalls and Common Mistakes

New and old users alike can run into a pitfall. Below we outline issues that we see frequently as well as explain how to resolve those issues. In the #nginx IRC channel on Libera Chat, we see these issues frequently.

This Guide Says

The most frequent issue we see happens when someone attempts to just copy and paste a configuration snippet from some other guide. Not all guides out there are wrong, but a scary number of them are.

These docs were created and reviewed by community members that work directly with all types of NGINX users. This specific document exists only because of the volume of common and recurring issues that community members see.

My Issue Isn't Listed

You don't see something in here related to your specific issue. Maybe we didn't point you here because of the exact issue you're experiencing. Don't skim this page and assume you were sent here for no reason. You were sent here because something you did wrong is listed here.

When it comes to supporting many users on many issues, community members don't want to support broken configurations. Fix your configuration before asking for help. Fix your configuration by reading through this. Don't just skim it.

Chmod 777

NEVER use `777`. It might be one nifty number, but even in testing it's a sign of having no clue what you're doing. Look at the permissions in the whole path and think through what's going on.

To easily display all the permissions on a path, you can use:

```
namei -om /path/to/check
```

Root inside Location Block

BAD:

```
server {
    server_name www.example.com;
    location / {
        root /var/www/nginx-default/;
        # [...]
      }
    location /foo {
        root /var/www/nginx-default/;
        # [...]
    }
    location /bar {
        root /some/other/place;
        # [...]
    }
}
```

This works. Putting root inside of a `location` block will work and it's perfectly valid. What's wrong is when you start adding `location` blocks. If you add a root to every location block then a `location` block that isn't matched will have no root. Therefore, it is important that a root directive occur prior to your `location` blocks, which can then override this directive if they need to. Let's look at a good configuration.

GOOD:

```
server {
    server_name www.example.com;
    root /var/www/nginx-default/;
    location / {
        # [...]
    }
    location /foo {
        # [...]
    }
    location /bar {
        root /some/other/place;
        # [...]
    }
}
```

Multiple Index Directives

BAD:

```
http {
    index index.php index.htm index.html;
    server {
        server_name www.example.com;
        location / {
            index index.php index.htm index.html;
            # [...]
        }
    }
    server {
        server_name example.com;
        location / {
            index index.php index.htm index.html;
            # [...]
        }
        location /foo {
            index index.php;
            # [...]
        }
    }
}
```

Why repeat so many lines when not needed? Simply use the `index` directive one time. It only needs to occur in your `http{}` block and it will be inherited below.

GOOD:

```
http {
    index index.php index.htm index.html;
    server {
        server_name www.example.com;
        location / {
            # [...]
        }
    }
    server {
        server_name example.com;
        location / {
            # [...]
        }
        location /foo {
            # [...]
        }
    }
}
```

Using `if`

There is a little page about using `if` statements. It's called IfIsEvil and you really should check it out. Let's take a look at a few uses of `if` that are bad.

See also:
If Is Evil

Server Name (If)

BAD:

```
server {
    server_name example.com *.example.com;
        if ($host ~* ^www\.(.+)) {
            set $raw_domain $1;
            rewrite ^/(.*)$ $raw_domain/$1 permanent;
        }
        # [...]
    }
}
```

There are actually three problems here. The first being the `if`. That's what we care about now. Why is this bad? Did you read If is Evil? When NGINX receives a request - no matter what is the subdomain being requested, be it www.example.com or just the plain example.com - this `if` directive is **always** evaluated. Since you're requesting NGINX to check for the Host header for **every request**, it's extremely inefficient. You should avoid it. Instead use two `server` directives like the example below.

GOOD:

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

Besides making the configuration file easier to read. This approach decreases NGINX processing requirements. We got rid of the spurious `if`. We're also using `$scheme` which doesn't hardcode the URI scheme you're using, be it http or https.

Check (If) File Exists

Using `if` to ensure a file exists is horrible. It's mean. If you have any recent version of NGINX you should look at `try_files` which just made life much easier.

BAD:

```
server {
    root /var/www/example.com;
    location / {
        if (!-f $request_filename) {
            break;
        }
    }
}
```

GOOD:

```
server {
    root /var/www/example.com;
    location / {
        try_files $uri $uri/ /index.html;
    }
}
```

What we changed is that we try to see if `$uri` exists without requiring `if`. Using `try_files` means that you can test a sequence. If `$uri` doesn't exist, try `$uri/`, if that doesn't exist try a fallback location.

In this case, if the `$uri` file exists, serve it. If not, check if that directory exists. If not, then proceed to serve `index.html` which you make sure exists. It's loaded – but oh-so-simple! This is another instance where you can completely eliminate If.

Front Controller Pattern Web Apps

"Front Controller Pattern" designs are popular and are used on the many of the most popular PHP software packages; But a lot of examples are more complex than they need to be. For Drupal, Joomla, etc., just use this:

```
try_files $uri $uri/ /index.php?q=$uri&$args;
```

Note - the parameter names are different based on the package you're using. For example:

- "q" is the parameter used by Drupal, Joomla, WordPress
- "page" is used by CMS Made Simple

Some software don't even need the query string and can read from REQUEST_URI. For example, WordPress supports this:

```
try_files $uri $uri/ /index.php;
```

If you don't care about checking for the existence of directories, you can skip it by removing `$uri/`.

Of course, your mileage may vary and you may require something more complex based on your needs, but for basic sites, these will work perfectly. You should always start simple and build from there.

Passing Uncontrolled Requests to PHP

Many example NGINX configurations for PHP on the web advocate passing every URI ending in `.php` to the PHP interpreter. Note that this presents a serious security issue on most PHP setups as it may allow arbitrary code execution by third parties.

The problem section usually looks like this:

```
location ~* \.php$ {
    fastcgi_pass backend;
    # [...]
}
```

Here, every request ending in `.php` will be passed to the FastCGI backend. The issue with this is that the default PHP configuration tries to guess which file you want to execute if the full path does not lead to an actual file on the filesystem.

For instance, if a request is made for */forum/avatar/1232.jpg/file.php* which does not exist but if */forum/avatar/1232.jpg* does, the PHP interpreter will process */forum/avatar/1232.jpg* instead. If this contains embedded PHP code, this code will be executed accordingly.

Options for avoiding this are:

- Set `cgi.fix_pathinfo=0` in `php.ini`. This causes the PHP interpreter to only try the literal path given and to stop processing if the file is not found.
- Ensure that NGINX only passes specific PHP files for execution:

```
location ~* (file_a|file_b|file_c)\.php$ {
    fastcgi_pass backend;
    # [...]
}
```

- Specifically disable the execution of PHP files in any directory containing user uploads:

```
location /uploaddir {
    location ~ \.php$ {return 403;}
    # [...]
}
```

- Use the `try_files` directive to filter out the problem condition:

```
location ~* \.php$ {
    try_files $uri =404;
    fastcgi_pass backend;
    # [...]
}
```

- Use a nested location to filter out the problem condition:

```
location ~* \.php$ {
    location ~ \..*/.*\.php$ {return 404;}
    fastcgi_pass backend;
    # [...]
}
```

FastCGI Path in Script Filename

So many guides out there like to rely on absolute paths to get to your information. This is commonly seen in PHP blocks. When you install NGINX from a repository, you'll usually wind up being able to toss `include fastcgi_params;` in your config. This is a file located in your NGINX root directory which is usually around `/etc/nginx/`.

GOOD:

```
fastcgi_param  SCRIPT_FILENAME    $document_root$fastcgi_script_name;
```

BAD:

```
fastcgi_param  SCRIPT_FILENAME    /var/www/yoursite.com/$fastcgi_script_nam
e;
```

Where is `$document_root` set? It's set by the root directive that should be in your server block. Is your root directive not there? See the first pitfall.

Taxing Rewrites

Don't feel bad here, it's easy to get confused with regular expressions. In fact, it's so easy to do that we should make an effort to keep them neat and clean. Quite simply, don't add cruft.

BAD:

```
rewrite ^/(.*)$ http://example.com/$1 permanent;
```

GOOD:

```
rewrite ^ http://example.com$request_uri? permanent;
```

BETTER:

```
return 301 http://example.com$request_uri;
```

Look at the above. Then back here. Then up, and back here. OK. The first rewrite captures the full URI minus the first slash. By using the built-in variable `$request_uri` we can effectively avoid doing any capturing or matching at all.

Rewrite Missing `http://`

Very simply, rewrites are relative unless you tell NGINX that they're not. Making a rewrite absolute is simple. Add a scheme.

BAD:

```
rewrite ^ example.com permanent;
```

GOOD:

```
rewrite ^ http://example.com permanent;
```

In the above you will see that all we did was add `http://` to the rewrite. It's simple, easy, and effective.

Proxy Everything

BAD:

```
server {
    server_name _;
    root /var/www/site;
    location / {
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_pass unix:/tmp/phpcgi.socket;
    }
}
```

Yucky. In this instance, you pass EVERYTHING to PHP. Why? Apache might do this, but you don't need to. The try_files directive exists for an amazing reason: It tries files in a specific order. NGINX can first try to serve the static content, and if it can't, it moves on. This means PHP doesn't get

involved at all. MUCH faster. Especially if you're serving a 1MB image over PHP a few thousand times versus serving it directly. Let's take a look at how to do that.

GOOD:

```
server {
    server_name _;
    root /var/www/site;
    location / {
        try_files $uri $uri/ @proxy;
    }
    location @proxy {
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_pass unix:/tmp/phpcgi.socket;
    }
}
```

Also GOOD:

```
server {
    server_name _;
    root /var/www/site;
    location / {
        try_files $uri $uri/ /index.php;
    }
    location ~ \.php$ {
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_pass unix:/tmp/phpcgi.socket;
    }
}
```

Easy, right? Check if the requested URI exists and can be served by NGINX. If not, check if it is a directory that can be served. If not, then pass it to your proxy. Only when NGINX can't serve that requested URI directly, your proxy overhead will get involved.

Consider how many of your requests are for static content (images, css, javascript, etc.). That's probably a lot of overhead you just saved.

Use `$request_filename` for `SCRIPT_FILENAME`

Use `$request_filename` instead of `$document_root$fastcgi_script_name`.

If you use the `alias` directive with `$document_root$fastcgi_script_name`, `$document_root$fastcgi_script_name` will return the wrong path.

BAD:

```
    location /api/ {
        index  index.php index.html index.htm;
        alias /app/www/;
        location ~* "\.php$" {
            try_files       $uri =404;
            fastcgi_pass    127.0.0.1:9000;
            fastcgi_index   index.php;
            fastcgi_param   SCRIPT_FILENAME  $document_root$fastcgi_script_name
    ;
        }
     }
```

Request `/api/testing.php`:

- `$document_root$fastcgi_script_name` == `/app/www//api/testing.php`
- `$request_filename` == `/app/www/testing.php`

Request `/api/`:

- `$document_root$fastcgi_script_name` == `/app/www//api/index.php`
- `$request_filename` == `/app/www/index.php`

And if you use `$request_filename`, you should set index using `index` directive, `fastcgi_index` will not work.

GOOD:

```
    location /api/ {
        index  index.php index.html index.htm;
        alias /app/www/;
        location ~* "\.php$" {
            try_files       $uri =404;
            fastcgi_pass    127.0.0.1:9000;
            fastcgi_param   SCRIPT_FILENAME  $request_filename;
        }
     }
```

Config Changes Not Reflected

Browser cache. Your configuration may be perfect but you'll sit there and beat your head against a cement wall for a month. What's wrong is your browser cache. When you download something, your browser stores it. It also stores how that file was served. If you are playing with a `types{}` block you'll encounter this.

The fix:

- In Firefox press Ctrl+Shift+Delete, check Cache, click Clear Now. In any other browser, just ask your favorite search engine. Do this after every change (unless you know it's not needed) and you'll save yourself a lot of headaches.
- Use curl.

VirtualBox

If this does not work, and you're running NGINX on a virtual machine in VirtualBox, it may be sendfile() that is causing the trouble. Simply comment out the sendfile directive or set it to "off". The directive is most likely found in your nginx.conf file.:

```
sendfile off;
```

Missing (disappearing) HTTP Headers

If you do not explicitly set `underscores_in_headers on;`, NGINX will silently drop HTTP headers with underscores (which are perfectly valid according to the HTTP standard). This is done in order to prevent ambiguities when mapping headers to CGI variables as both dashes and underscores are mapped to underscores during that process.

Not Using Standard Document Root Locations

Some directories in any file system should never be used for hosting data from. These include `/` and `root`. You should never use these as your document root.

Doing this leaves you open to a request outside of your expected area returning private data.

NEVER DO THIS!!! (yes, we have seen this)

```
server {
    root /;

    location / {
        try_files /web/$uri $uri @php;
    }

    location @php {
        # [...]
    }
}
```

When a request is made for `/foo`, the request is passed to php because the file isn't found. This can appear fine, until a request in made for `/etc/passwd`. Yup, you just gave us a list of all users on that server. In some cases, the NGINX server is even set up run workers as root. Yup, we now have your user list as well as password hashes and how they've been hashed. We now own your box.

The File System Hierarchy defines where data should exist. You should definitely read it. The short version is that you want your web content to exist in either `/var/www/`, `/srv`, `/usr/share/www`.

Using the Default Document Root

NGINX packages that exist in Ubuntu, Debian, or other operating systems, as an easy-to-install package will often provide a 'default' configuration file as an example of configuration methods, and will often include a document root to hold a basic HTML file.

Most of these packaging systems do not check to see if files are modified or exist within the default document root, which can result in code loss when the packages are upgraded. Experienced system administrators know that there is no expectation of the data inside the default document root to remain untouched during upgrades.

You should not use the default document root for any site-critical files. There is no expectation that the default document root will be left untouched by the system and there is an extremely high possibility that your site-critical data may be lost upon updates and upgrades to the NGINX packages for your operating system.

Using a Hostname to Resolve Addresses

BAD:

```
upstream {
    server http://someserver;
}

server {
    listen myhostname:80;
    # [...]
}
```

You should never use a hostname in a listen directive. While this may work, it will come with a large number of issues. One such issue being that the hostname may not resolve at boot time or during a service restart. This can cause NGINX to be unable to bind to the desired TCP socket which will prevent NGINX from starting at all.

A safer practice is to know the IP address that needs to be bound to and use that address instead of the hostname. This prevents NGINX from needing to look up the address and removes

dependencies on external and internal resolvers.

This same issue applies to upstream locations. While it may not always be possible to avoid using a hostname in an upstream block, it is bad practice and will require careful considerations to prevent issues.

GOOD:

```
upstream {
    server http://10.48.41.12;
}

server {
    listen 127.0.0.16:80;
    # [...]
}
```

Using SSLv3 with HTTPS

Due to the POODLE vulnerability in SSLv3, it is advised to not use SSLv3 in your SSL-enabled sites. You can very easily disable SSLv3 with this line and provide only the TLS protocols instead:

```
ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
```

Using the `try_files $uri` directive with `alias`

The symptoms of this are difficult to diagnose: typically, it will appear that you've done everything right and yet you get mysterious 404 errors. Why? well, turning on debug-level error logging reveals that `try_files` is appending `$uri` onto the path already set with `alias`. This is due to a bug in NGINX, but don't worry—the workaround is simple! As long as your `try_files` line is something like `try_files $uri $uri/ =404;`, you can simply delete the `try_files` line with no significant adverse effect. Here is an example where you cannot use `try_files`.

BAD:

```
location ~^/\~(?<user>[^/]*)/(?<page>.*)$ {
    alias /home/$user/public_html/$page;
    try_files $uri $uri/ =404;
}
```

GOOD:

```
location ~^/\~(?<user>[^/]*)/(?<page>.*)$ {
    alias /home/$user/public_html/$page;
}
```

The one caveat is that this workaround prevents you from using `try_files` to avoid PATH_INFO attacks. See Passing Uncontrolled Requests to PHP above for alternative ways to mitigate these attacks.

Also note that the `snippets/fastcgi-php.conf` file shipped by some Linux distributions may need to be edited to remove a `try_files` directive if it's included in a `location` block with `alias`.

Incorrect `return` context

The return directive applies only inside the topmost context it's defined in. In this example:

```
server {
    location /a/ {
        try_files test.html =404;
    }

    return 301 http://example.org;
}
```

A request to `/a/test.html` will return a 301. To make this work as expected wrap the second block inside a `location /`:

```
server {
    location /a/ {
        try_files test.html =404;
    }

    location / {
        return 301 http://example.org;
    }
}
```