Serving Static Content

Configure NGINX and NGINX Plus to serve static content, with type-specific root directories, checks for file existence, and performance optimizations.

This section describes how to configure NGINX and NGINX Plus to serve static content, how to define which paths are searched to find requested files, how to set up index files, and how to tune NGINX and NGINX Plus, as well as the kernel, for optimal performance.

Root Directory and Index Files

The root directive specifies the root directory that will be used to search for a file. To obtain the path of a requested file, NGINX appends the request URI to the path specified by the `root` directive. The directive can be placed on any level within the `http {}`, `server {}`, or `location {}` contexts. In the example below, the `root` directive is defined for a virtual server. It applies to all `location {}` blocks where the `root` directive is not included to explicitly redefine the root.

```
server {
    root /www/data;

    location / {
    }

    location /images/ {
    }

    location ~ \.(mp3|mp4) {
        root /www/media;
    }
}
```

Here, NGINX searches for a URI that starts with `/images/` in the `/www/data/images/` directory in the file system. But if the URI ends with the `.mp3` or `.mp4` extension, NGINX instead searches for the file in the `/www/media/` directory because it is defined in the matching `location` block.

If a request ends with a slash, NGINX treats it as a request for a directory and tries to find an index file in the directory. The index directive defines the index file's name (the default value is `index.html`). To continue with the example, if the request URI is `/images/some/path/`, NGINX delivers the file `/www/data/images/some/path/index.html` if it exists. If it does not, NGINX returns HTTP code `404 (Not Found)` by default. To configure NGINX to return an automatically generated directory listing instead, include the `on` parameter to the autoindex directive:

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

You can list more than one filename in the `index` directive. NGINX searches for files in the specified order and returns the first one it finds.

```
location / {
    index index.$geo.html index.htm index.html;
}
```

The `$geo` variable used here here is a custom variable set through the geo directive. The value of the variable depends on the client's IP address.

To return the index file, NGINX checks for its existence and then makes an internal redirect to the URI obtained by appending the name of the index file to the base URI. The internal redirect results in a new search of a location and can end up in another location as in the following example:

```
location / {
    root /data;
    index index.html index.php;
}

location ~ \.php {
    fastcgi_pass localhost:8000;
    #...
}
```

Here, if the URI in a request is `/path/`, and `/data/path/index.html` does not exist but `/data/path/index.php` does, the internal redirect to `/path/index.php` is mapped to the second location. As a result, the request is proxied.

Trying Several Options

The try_files directive can be used to check whether the specified file or directory exists; NGINX makes an internal redirect if it does, or returns a specified status code if it doesn't. For example, to check the existence of a file corresponding to the request URI, use the `try_files` directive and the `$uri` variable as follows:

```
server {
    root /www/data;

    location /images/ {
        try_files $uri /images/default.gif;
    }
}
```

The file is specified in the form of the URI, which is processed using the `root` or `alias` directives set in the context of the current location or virtual server. In this case, if the file corresponding to the original URI doesn't exist, NGINX makes an internal redirect to the URI specified by the last parameter, returning `/www/data/images/default.gif`.

The last parameter can also be a status code (directly preceded by the equals sign) or the name of a location. In the following example, a `404` error is returned if none of the parameters to the `try_files` directive resolve to an existing file or directory.

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

In the next example, if neither the original URI nor the URI with the appended trailing slash resolve into an existing file or directory, the request is redirected to the named location which passes it to a proxied server.

```
location / {
    try_files $uri $uri/ @backend;
}

location @backend {
    proxy_pass http://backend.example.com;
}
```

For more information, watch the Content Caching webinar on-demand to learn how to dramatically improve the performance of a website, and get a deep-dive into NGINX's caching capabilities.

Optimizing Performance for Serving Content

Loading speed is a crucial factor of serving any content. Making minor optimizations to your NGINX configuration may boost the productivity and help reach optimal performance.

Enabling `sendfile`

By default, NGINX handles file transmission itself and copies the file into the buffer before sending it. Enabling the sendfile directive eliminates the step of copying the data into the buffer and enables direct copying data from one file descriptor to another. Alternatively, to prevent one fast connection from entirely occupying the worker process, you can use the sendfile_max_chunk directive to limit the amount of data transferred in a single `sendfile()` call (in this example, to `1` MB):

```
location /mp3 {
    sendfile           on;
    sendfile_max_chunk 1m;
    #...
}
```

Enabling `tcp_nopush`

Use the tcp_nopush directive together with the sendfile `on` directive. This enables NGINX to send HTTP response headers in one packet right after the chunk of data has been obtained by `sendfile()`.

```
location /mp3 {
    sendfile   on;
    tcp_nopush on;
    #...
}
```

Enabling `tcp_nodelay`

The tcp_nodelay directive allows override of Nagle's algorithm, originally designed to solve problems with small packets in slow networks. The algorithm consolidates a number of small packets into a larger one and sends the packet with a `200` ms delay. Nowadays, when serving large static files, the data can be sent immediately regardless of the packet size. The delay also affects online applications (ssh, online games, online trading, and so on). By default, the tcp_nodelay directive is set to `on` which means that the Nagle's algorithm is disabled. Use this directive only for keepalive connections:

```
location /mp3  {
    tcp_nodelay       on;
    keepalive_timeout 65;
    #...
}
```

Optimizing the Backlog Queue

One of the important factors is how fast NGINX can handle incoming connections. The general rule is when a connection is established, it is put into the "listen" queue of a listen socket. Under normal load, either the queue is small or there is no queue at all. But under high load, the queue can grow dramatically, resulting in uneven performance, dropped connections, and increased latency.

Displaying the Listen Queue

To display the current listen queue, run this command:

```
netstat -Lan
```

The output might be like the following, which shows that in the listen queue on port `80` there are `10` unaccepted connections against the configured maximum of `128` queued connections. This situation is normal.

```
Current listen queue sizes (qlen/incqlen/maxqlen)
Listen         Local Address
0/0/128        *.12345
10/0/128       *.80
0/0/128        *.8080
```

In contrast, in the following command the number of unaccepted connections (`192`) exceeds the limit of `128`. This is quite common when a web site experiences heavy traffic. To achieve optimal performance, you need to increase the maximum number of connections that can be queued for acceptance by NGINX in both your operating system and the NGINX configuration.

```
Current listen queue sizes (qlen/incqlen/maxqlen)
Listen         Local Address
0/0/128        *.12345
192/0/128        *.80
0/0/128        *.8080
```

Tuning the Operating System

Increase the value of the `net.core.somaxconn` kernel parameter from its default value (`128`) to a value high enough for a large burst of traffic. In this example, it's increased to `4096`.

- For FreeBSD, run the command:

  ```
  sudo sysctl kern.ipc.somaxconns=4096
  ```

- For Linux:

  1. Run the command:

     ```
     sudo sysctl -w net.core.somaxconn=4096
     ```

  2. Use a text editor to add the following line to `/etc/sysctl.conf`

     ```
     net.core.somaxconn = 4096
     ```

Tuning NGINX

If you set the `somaxconn` kernel parameter to a value greater than `512`, change the `backlog` parameter to the NGINX listen directive to match:

```
server {
    listen 80 backlog=4096;
    # ...
}
```