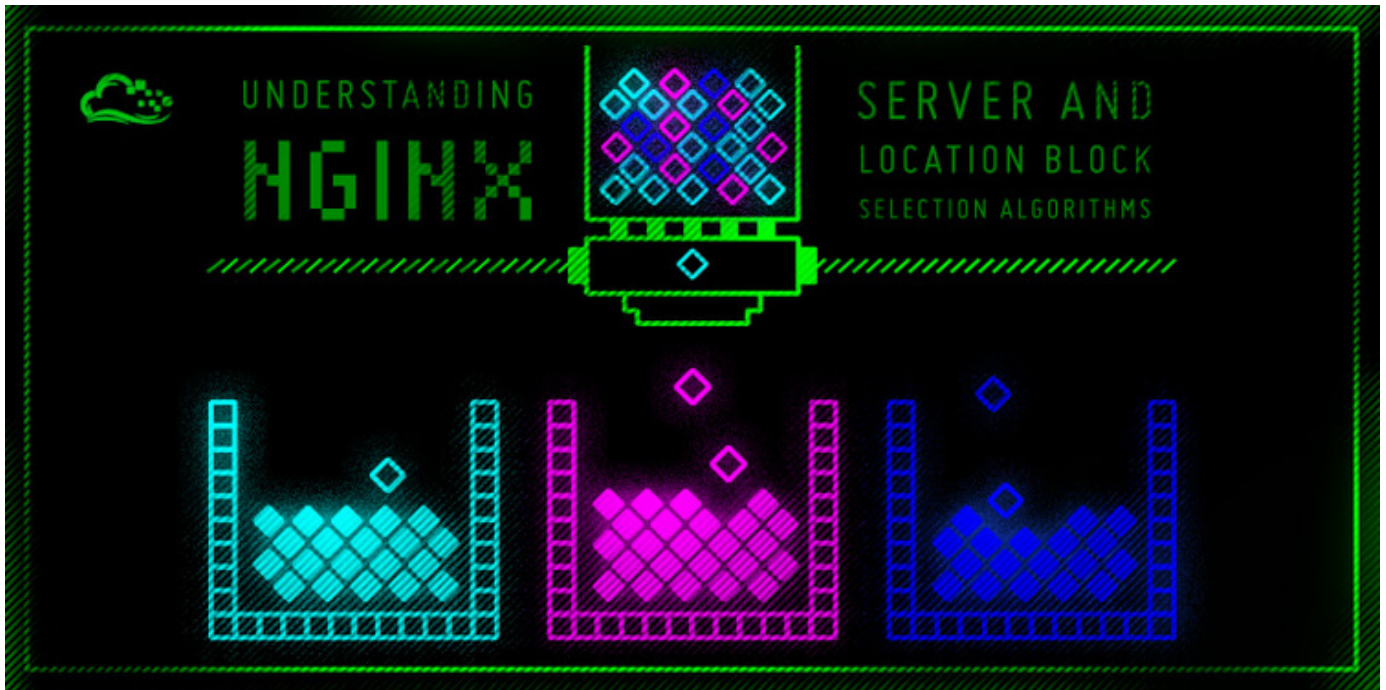# Understanding Nginx Server and Location Block Selection Algorithms

Posted Nov 17, 2014 263k
views

Introduction

Nginx is one of the most popular web servers in the world. It can successfully handle high loads with many concurrent client connections, and can easily function as a web server, a mail server, or a reverse proxy server.

In this guide, we will discuss some of the behind-the-scenes details that determine how Nginx processes client requests. Understanding these ideas can help take the guesswork out of designing server and location blocks and can make the request handling seem less unpredictable.

Nginx Block Configurations

Nginx logically divides the configurations meant to serve different content into blocks, which live in a hierarchical structure. Each time a client request is made, Nginx begins a process of determining which configuration blocks should be used to handle the request. This decision process is what we will be discussing in this guide.

The main blocks that we will be discussing are the **server** block and the **location** block.

A server block is a subset of Nginx's configuration that defines a virtual server used to handle requests of a defined type. Administrators often configure multiple server blocks and decide which block should handle which connection based on the requested domain name, port, and IP address.

A location block lives within a server block and is used to define how Nginx should handle requests for different resources and URIs for the parent server. The URI space can be subdivided in whatever way the administrator likes using these blocks. It is an extremely flexible model.

How Nginx Decides Which Server Block Will Handle a Request

Since Nginx allows the administrator to define multiple server blocks that function as separate virtual web server instances, it needs a procedure for determining which of these server blocks will be used to satisfy a request.

It does this through a defined system of checks that are used to find the best possible match. The main server block directives that Nginx is concerned with during this process are the `listen` directive, and the `server_name` directive.

Parsing the "listen" Directive to Find Possible Matches

First, Nginx looks at the IP address and the port of the request. It matches this against the `listen` directive of each server to build a list of the server blocks that can possibly resolve the request.

The `listen` directive typically defines which IP address and port that the server block will respond to. By default, any server block that does not include a `listen` directive is given the listen parameters of `0.0.0.0:80` (or `0.0.0.0:8080` if Nginx is being run by a normal, non-root user). This allows these blocks to respond to requests on any interface on port 80, but this default value does not hold much weight within the server selection process.

The `listen` directive can be set to:

- An IP address/port combo.
- A lone IP address which will then listen on the default port 80.
- A lone port which will listen to every interface on that port.
- The path to a Unix socket.

The last option will generally only have implications when passing requests between different servers.

When trying to determine which server block to send a request to, Nginx will first try to decide based on the specificity of the `listen` directive using the following rules:

- Nginx translates all "incomplete" `listen` directives by substituting missing values with their default values so that each block can be evaluated by its IP address and port. Some examples of these translations are:
  - A block with no `listen` directive uses the value `0.0.0.0:80`.
  - A block set to an IP address `111.111.111.111` with no port becomes `111.111.111.111:80`
  - A block set to port `8888` with no IP address becomes `0.0.0.0:8888`

- Nginx then attempts to collect a list of the server blocks that match the request most specifically based on the IP address and port. This means that any block that is functionally using `0.0.0.0` as its IP address (to match any interface), will not be selected if there are matching blocks that list a specific IP address. In any case, the port must be matched exactly.

- If there is only one most specific match, that server block will be used to serve the request. If there are multiple server blocks with the same level of specificity matching, Nginx then begins to evaluate the `server_name` directive of each server block.

It is important to understand that Nginx will only evaluate the `server_name` directive when it needs to distinguish between server blocks that match to the same level of specificity in the `listen` directive. For instance, if `example.com` is hosted on port `80` of `192.168.1.10`, a request for `example.com` will always be served by the first block in this example, despite the `server_name` directive in the second block.

```
server {

    listen 192.168.1.10;


    . . .


}

server {

    listen 80;
    server_name example.com;


    . . .


}
```

In the event that more than one server block matches with equal specificity, the next step is to check the `server_name` directive.

Parsing the "server_name" Directive to Choose a Match

Next, to further evaluate requests that have equally specific `listen` directives, Nginx checks the request's "Host" header. This value holds the domain or IP address that the client was actually trying to reach.

Nginx attempts to find the best match for the value it finds by looking at the `server_name` directive within each of the server blocks that are still selection candidates. Nginx evaluates these by using the following formula:

- Nginx will first try to find a server block with a `server_name` that matches the value in the "Host" header of the request *exactly*. If this is found, the associated block will be used to serve the request. If multiple exact matches are found, the **first** one is used.
- If no exact match is found, Nginx will then try to find a server block with a `server_name` that matches using a leading wildcard (indicated by a `*` at the beginning of the name in the config). If one is found, that block will be used to serve the request. If multiple matches are found, the **longest** match will be used to serve the request.
- If no match is found using a leading wildcard, Nginx then looks for a server block with a `server_name` that matches using a trailing wildcard (indicated by a server name ending with a `*` in the config). If one is found, that block is used to serve the request. If multiple matches are found, the **longest** match will be used to serve the request.

- If no match is found using a trailing wildcard, Nginx then evaluates server blocks that define the `server_name` using regular expressions (indicated by a `~` before the name). The **first** `server_name` with a regular expression that matches the "Host" header will be used to serve the request.
- If no regular expression match is found, Nginx then selects the default server block for that IP address and port.

Each IP address/port combo has a default server block that will be used when a course of action can not be determined with the above methods. For an IP address/port combo, this will either be the first block in the configuration or the block that contains the `default_server` option as part of the `listen` directive (which would override the first-found algorithm). There can be only one `default_server` declaration per each IP address/port combination.

Examples

If there is a `server_name` defined that exactly matches the "Host" header value, that server block is selected to process the request.

In this example, if the "Host" header of the request was set to "host1.example.com", the second server would be selected:

```
server {
    listen 80;
    server_name *.example.com;

    . . .

}

server {
    listen 80;
    server_name host1.example.com;

    . . .

}
```

If no exact match is found, Nginx then checks to see if there is a `server_name` with a starting wildcard that fits. The longest match beginning with a wildcard will be selected to fulfill the request.

In this example, if the request had a "Host" header of "www.example.org", the second server block would be selected:

```
server {
    listen 80;
    server_name www.example.*;

    . . .
```

```
}

server {
    listen 80;
    server_name *.example.org;

    . . .

}

server {
    listen 80;
    server_name *.org;

    . . .

}
```

If no match is found with a starting wildcard, Nginx will then see if a match exists using a wildcard at the end of the expression. At this point, the longest match ending with a wildcard will be selected to serve the request.

For instance, if the request has a "Host" header set to "www.example.com", the third server block will be selected:

```
server {
    listen 80;
    server_name host1.example.com;

    . . .

}

server {
    listen 80;
    server_name example.com;

    . . .

}

server {
    listen 80;
    server_name www.example.*;

    . . .
```

```
}
```

If no wildcard matches can be found, Nginx will then move on to attempting to match `server_name` directives that use regular expressions. The *first* matching regular expression will be selected to respond to the request.

For example, if the "Host" header of the request is set to "www.example.com", then the second server block will be selected to satisfy the request:

```
server {
    listen 80;
    server_name example.com;

    . . .

}

server {
    listen 80;
    server_name ~^(www|host1).*\.example\.com$;

    . . .

}

server {
    listen 80;
    server_name ~^(subdomain|set|www|host1).*\.example\.com$;

    . . .

}
```

If none of the above steps are able to satisfy the request, then the request will be passed to the *default* server for the matching IP address and port.

Matching Location Blocks

Similar to the process that Nginx uses to select the server block that will process a request, Nginx also has an established algorithm for deciding which location block within the server to use for handling requests.

Location Block Syntax

Before we cover how Nginx decides which location block to use to handle requests, let's go over some of the syntax you might see in location block definitions. Location blocks live within server blocks (or other location blocks) and are used to decide how to process the request URI (the part of the request that comes after the domain name or IP address/port).

Location blocks generally take the following form:

```
location optional_modifier location_match {

    . . .

}
```

The `location_match` in the above defines what Nginx should check the request URI against. The existence or nonexistence of the modifier in the above example affects the way that the Nginx attempts to match the location block. The modifiers below will cause the associated location block to be interpreted as follows:

- **(none)**: If no modifiers are present, the location is interpreted as a *prefix* match. This means that the location given will be matched against the beginning of the request URI to determine a match.
- `=`: If an equal sign is used, this block will be considered a match if the request URI exactly matches the location given.
- `~`: If a tilde modifier is present, this location will be interpreted as a case-sensitive regular expression match.
- `~*`: If a tilde and asterisk modifier is used, the location block will be interpreted as a case-insensitive regular expression match.
- `^~`: If a carat and tilde modifier is present, and if this block is selected as the best non-regular expression match, regular expression matching will not take place.

Examples Demonstrating Location Block Syntax

As an example of prefix matching, the following location block may be selected to respond for request URIs that look like `/site`, `/site/page1/index.html`, or `/site/index.html`:

```
location /site {

    . . .

}
```

For a demonstration of exact request URI matching, this block will always be used to respond to a request URI that looks like `/page1`. It will **not** be used to respond to a `/page1/index.html` request URI. Keep in mind that if this block is selected and the request is fulfilled using an index page, an internal redirect will take place to another location that will be the actual handler of the request:

```
location = /page1 {

    . . .

}
```

As an example of a location that should be interpreted as a case-sensitive regular expression, this block could be used to handle requests for `/tortoise.jpg`, but **not** for `/FLOWER.PNG`:

```
location ~ \.(jpe?g|png|gif|ico)$ {

    . . .

}
```

A block that would allow for case-insensitive matching similar to the above is shown below. Here, both `/tortoise.jpg` *and* `/FLOWER.PNG` could be handled by this block:

```
location ~* \.(jpe?g|png|gif|ico)$ {

    . . .

}
```

Finally, this block would prevent regular expression matching from occurring if it is determined to be the best non-regular expression match. It could handle requests for `/costumes/ninja.html`:

```
location ^~ /costumes {

    . . .

}
```

As you see, the modifiers indicate how the location block should be interpreted. However, this does *not* tell us the algorithm that Nginx uses to decide which location block to send the request to. We will go over that next.

How Nginx Chooses Which Location to Use to Handle Requests

Nginx chooses the location that will be used to serve a request in a similar fashion to how it selects a server block. It runs through a process that determines the best location block for any given request. Understanding this process is a crucial requirement in being able to configure Nginx reliably and accurately.

Keeping in mind the types of location declarations we described above, Nginx evaluates the possible location contexts by comparing the request URI to each of the locations. It does this using the following algorithm:

- Nginx begins by checking all prefix-based location matches (all location types not involving a regular expression). It checks each location against the complete request URI.
- First, Nginx looks for an exact match. If a location block using the `=` modifier is found to match the request URI exactly, this location block is immediately selected to serve the request.
- If no exact (with the `=` modifier) location block matches are found, Nginx then moves on to evaluating non-exact prefixes. It discovers the longest matching prefix location for the given request URI, which it then evaluates as follows:

- If the longest matching prefix location has the `^~` modifier, then Nginx will immediately end its search and select this location to serve the request.
- If the longest matching prefix location *does not* use the `^~` modifier, the match is stored by Nginx for the moment so that the focus of the search can be shifted.

- After the longest matching prefix location is determined and stored, Nginx moves on to evaluating the regular expression locations (both case sensitive and insensitive). If there are any regular expression locations *within* the longest matching prefix location, Nginx will move those to the top of its list of regex locations to check. Nginx then tries to match against the regular expression locations sequentially. The **first** regular expression location that matches the request URI is immediately selected to serve the request.
- If no regular expression locations are found that match the request URI, the previously stored prefix location is selected to serve the request.

It is important to understand that, by default, Nginx will serve regular expression matches in preference to prefix matches. However, it *evaluates* prefix locations first, allowing for the administer to override this tendency by specifying locations using the `=` and `^~` modifiers.

It is also important to note that, while prefix locations generally select based on the longest, most specific match, regular expression evaluation is stopped when the first matching location is found. This means that positioning within the configuration has vast implications for regular expression locations.

Finally, it it is important to understand that regular expression matches *within* the longest prefix match will "jump the line" when Nginx evaluates regex locations. These will be evaluated, in order, before any of the other regular expression matches are considered. Maxim Dounin, an incredibly helpful Nginx developer, explains in this post this portion of the selection algorithm.

When Does Location Block Evaluation Jump to Other Locations?

Generally speaking, when a location block is selected to serve a request, the request is handled entirely within that context from that point onward. Only the selected location and the inherited directives determine how the request is processed, without interference from sibling location blocks.

Although this is a general rule that will allow you to design your location blocks in a predictable way, it is important to realize that there are times when a new location search is triggered by certain directives within the selected location. The exceptions to the "only one location block" rule may have implications on how the request is actually served and may not align with the expectations you had when designing your location blocks.

Some directives that can lead to this type of internal redirect are:

- **index**
- **try_files**
- **rewrite**
- **error_page**

Let's go over these briefly.

The `index` directive always leads to an internal redirect if it is used to handle the request. Exact location matches are often used to speed up the selection process by immediately ending the execution of the algorithm. However, if you make an exact location match that is a *directory*, there is a good chance that the request will be redirected to a different location for actual processing.

In this example, the first location is matched by a request URI of `/exact`, but in order to handle the request, the `index` directive inherited by the block initiates an internal redirect to the second block:

```
index index.html;

location = /exact {

    . . .

}

location / {

    . . .

}
```

In the case above, if you really need the execution to stay in the first block, you will have to come up with a different method of satisfying the request to the directory. For instance, you could set an invalid `index` for that block and turn on `autoindex`:

```
location = /exact {
    index nothing_will_match;
    autoindex on;
}

location  / {

    . . .

}
```

This is one way of preventing an `index` from switching contexts, but it's probably not useful for most configurations. Mostly an exact match on directories can be helpful for things like rewriting the request (which also results in a new location search).

Another instance where the processing location may be reevaluated is with the `try_files` directive. This directive tells Nginx to check for the existence of a named set of files or directories. The last parameter can be a URI that Nginx will make an internal redirect to.

Consider the following configuration:

```
root /var/www/main;
location / {
    try_files $uri $uri.html $uri/ /fallback/index.html;
}


location /fallback {
    root /var/www/another;
}
```

In the above example, if a request is made for `/blahblah`, the first location will initially get the request. It will try to find a file called `blahblah` in `/var/www/main` directory. If it cannot find one, it will follow up by searching for a file called `blahblah.html`. It will then try to see if there is a directory called `blahblah/` within the `/var/www/main` directory. Failing all of these attempts, it will redirect to `/fallback/index.html`. This will trigger another location search that will be caught by the second location block. This will serve the file `/var/www/another/fallback/index.html`.

Another directive that can lead to a location block pass off is the `rewrite` directive. When using the `last` parameter with the `rewrite` directive, or when using no parameter at all, Nginx will search for a new matching location based on the results of the rewrite.

For example, if we modify the last example to include a rewrite, we can see that the request is sometimes passed directly to the second location without relying on the `try_files` directive:

```
root /var/www/main;
location / {
    rewrite ^/rewriteme/(.*)$ /$1 last;
    try_files $uri $uri.html $uri/ /fallback/index.html;
}


location /fallback {
    root /var/www/another;
}
```

In the above example, a request for `/rewriteme/hello` will be handled initially by the first location block. It will be rewritten to `/hello` and a location will be searched. In this case, it will match the first location again and be processed by the `try_files` as usual, maybe kicking back to `/fallback/index.html` if nothing is found (using the `try_files` internal redirect we discussed above).

However, if a request is made for `/rewriteme/fallback/hello`, the first block again will match. The rewrite be applied again, this time resulting in `/fallback/hello`. The request will then be served out of the second location block.

A related situation happens with the `return` directive when sending the `301` or `302` status codes. The difference in this case is that it results in an entirely new request in the form of an externally visible redirect. This same situation can occur with the `rewrite` directive when using the `redirect` or `permanent` flags.

However, these location searches shouldn't be unexpected, since externally visible redirects always result in a new request.

The `error_page` directive can lead to an internal redirect similar to that created by `try_files`. This directive is used to define what should happen when certain status codes are encountered. This will likely never be executed if `try_files` is set, since that directive handles the entire life cycle of a request.

Consider this example:

```
root /var/www/main;

location / {
    error_page 404 /another/whoops.html;
}

location /another {
    root /var/www;
}
```

Every request (other than those starting with `/another`) will be handled by the first block, which will serve files out of `/var/www/main`. However, if a file is not found (a 404 status), an internal redirect to `/another/whoops.html` will occur, leading to a new location search that will eventually land on the second block. This file will be served out of `/var/www/another/whoops.html`.

As you can see, understanding the circumstances in which Nginx triggers a new location search can help to predict the behavior you will see when making requests.

Conclusion

Understanding the ways that Nginx processes client requests can make your job as an administrator much easier. You will be able to know which server block Nginx will select based on each client request. You will also be able to tell how the location block will be selected based on the request URI. Overall, knowing the way that Nginx selects different blocks will give you the ability to trace the contexts that Nginx will apply in order to serve each request.