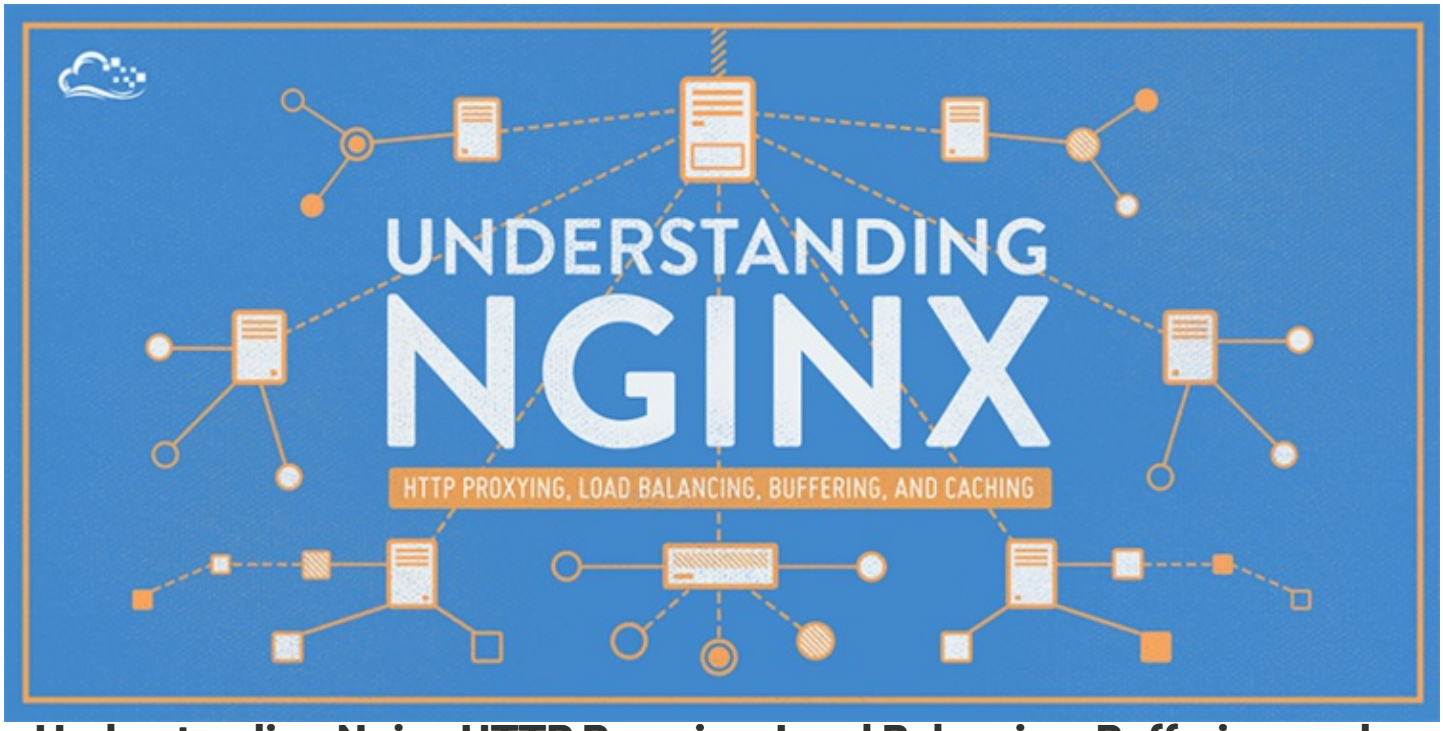


Understanding Nginx HTTP Proxying, Load Balancing, Buffering, and Caching

Posted November 25, 2014 728.7k SCALING CACHING CONCEPTUAL LOAD BALANCING NGINX

By Justin Ellingwood
Become an author

Introduction

In this guide, we will discuss Nginx's http proxying capabilities, which allow Nginx to pass requests off to backend http servers for further processing. Nginx is often set up as a reverse proxy solution to help scale out infrastructure or to pass requests to other servers that are not designed to handle large client loads.

Along the way, we will discuss how to scale out using Nginx's built-in load balancing capabilities. We will also explore buffering and caching to improve the performance of proxying operations for clients.

General Proxying Information

If you have only used web servers in the past for simple, single server configurations, you may be wondering why you would need to proxy requests.

One reason to proxy to other servers from Nginx is the ability to scale out your infrastructure. Nginx is built to handle many concurrent connections at the same time. This makes it ideal for being the point-of-contact for clients. The server can pass requests to any number of backend servers to handle the bulk of the work, which spreads the load across your infrastructure. This design also provides you with flexibility in easily adding backend servers or taking them down as needed for maintenance.

Another instance where an http proxy might be useful is when using an application servers that might not be built to handle requests directly from clients in production environments. Many frameworks include web servers, but most of them are not as robust as servers designed for high performance like Nginx. Putting Nginx in front of these servers can lead to a better experience for users and increased security.

Proxying in Nginx is accomplished by manipulating a request aimed at the Nginx server and passing it to other servers for the actual processing. The result of the request is passed back to Nginx, which then relays the information to the client. The other servers in this instance can be remote machines, local servers, or even other virtual servers defined within Nginx. The servers that Nginx proxies requests to are known as *upstream servers*.

Nginx can proxy requests to servers that communicate using the http(s), FastCGI, SCGI, and uwsgi, or memcached protocols through separate sets of directives for each type of proxy. In this guide, we will be focusing on the http protocol. The Nginx instance is responsible for passing on the request and massaging any message components into a format that the upstream server can understand.

Deconstructing a Basic HTTP Proxy Pass

The most straight-forward type of proxy involves handing off a request to a single server that can communicate using http. This type of proxy is known as a generic "proxy pass" and is handled by the aptly named `proxy_pass` directive.

The `proxy_pass` directive is mainly found in location contexts. It is also valid in `if` blocks within a location context and in `limit_except` contexts. When a request matches a location with a `proxy_pass` directive inside, the request is forwarded to the URL given by the directive.

Let's take a look at an example:

```
# server context

location /match/here {
    proxy_pass http://example.com;
}

. . .
```

In the above configuration snippet, no URI is given at the end of the server in the `proxy_pass` definition. For definitions that fit this pattern, the URI requested by the client will be passed to the upstream server as-is.

For example, when a request for `/match/here/please` is handled by this block, the request URI will be sent to the `example.com` server as `http://example.com/match/here/please`.

Let's take a look at the alternative scenario:

```
# server context

location /match/here {
    proxy_pass http://example.com/new/prefix;
}

. . .
```

In the above example, the proxy server is defined with a URI segment on the end (`/new/prefix`). When a URI is given in the `proxy_pass` definition, the portion of the request that matches the *location* definition is replaced by this URI during the pass.

For example, a request for `/match/here/please` on the Nginx server will be passed to the upstream server as `http://example.com/new/prefix/please`. The `/match/here` is replaced by `/new/prefix`. This is an important point to keep in mind.

Sometimes, this kind of replacement is impossible. In these cases, the URI at the end of the `proxy_pass` definition is ignored and either the original URI from the client or the URI as modified by other directives will be passed to the upstream server.

For instance, when the location is matched using regular expressions, Nginx cannot determine which part of the URI matched the expression, so it sends the original client request URI. Another example is when a rewrite directive is used within the same location, causing the client URI to be rewritten, but still handled in the same block. In this case, the rewritten URI will be passed.

Understanding How Nginx Processes Headers

One thing that might not be immediately clear is that it is important to pass more than just the URI if you expect the upstream server handle the request properly. The request coming from Nginx on behalf of a client will look different than a request coming directly from a client. A big part of this is the headers that go along with the request.

When Nginx proxies a request, it automatically makes some adjustments to the request headers it receives from the client:

- Nginx gets rid of any empty headers. There is no point of passing along empty values to another server; it would only serve to bloat the request.
- Nginx, by default, will consider any header that contains underscores as invalid. It will remove these from the proxied request. If you wish to have Nginx interpret these as valid, you can set the `underscores_in_headers` directive to "on", otherwise your headers will never make it to the backend server.
- The "Host" header is re-written to the value defined by the `$proxy_host` variable. This will be the IP address or name and port number of the upstream, directly as defined by the `proxy_pass` directive.
- The "Connection" header is changed to "close". This header is used to signal information about the particular connection established between two parties. In this instance, Nginx sets this to "close" to indicate to the upstream server that this connection will be closed once the original request is responded to. The upstream should not expect this connection to be persistent.

The first point that we can extrapolate from the above is that any header that you *do not* want passed should be set to an empty string. Headers with empty values are completely removed from the passed request.

The next point to glean from the above information is that if your backend application will be processing non-standard headers, you must make sure that they *do not* have underscores. If you need headers that use an underscore, you can set the `underscores_in_headers` directive to "on" further up in your configuration (valid either in the http context or in the context of the default server declaration for the IP address/port combination). If you do not do this, Nginx will flag these headers as invalid and silently drop them before passing to your upstream.

The "Host" header is of particular importance in most proxying scenarios. As stated above, by default, this will be set to the value of `$proxy_host`, a variable that will contain the domain name or IP address and port taken directly from the `proxy_pass` definition. This is selected by default as it is the only address Nginx can be sure the upstream server responds to (as it is pulled directly from the connection info).

The most common values for the "Host" header are below:

- `$proxy_host`: This sets the "Host" header to the domain name or IP address and port combo taken from the `proxy_pass` definition. This is the default and "safe" from Nginx's perspective, but not usually what is needed by the proxied server to correctly handle the request.

- `$http_host`: Sets the "Host" header to the "Host" header from the client request. The headers sent by the client are always available in Nginx as variables. The variables will start with an `$http_` prefix, followed by the header name in lowercase, with any dashes replaced by underscores. Although the `$http_host` variable works most of the time, when the client request does not have a valid "Host" header, this can cause the pass to fail.
- `$host`: This variable is set, in order of preference to: the host name from the request line itself, the "Host" header from the client request, or the server name matching the request.

In most cases, you will want to set the "Host" header to the `$host` variable. It is the most flexible and will usually provide the proxied servers with a "Host" header filled in as accurately as possible.

Setting or Resetting Headers

To adjust or set headers for proxy connections, we can use the `proxy_set_header` directive. For instance, to change the "Host" header as we have discussed, and add some additional headers common with proxied requests, we could use something like this:

```
# server context

location /match/here {
    proxy_set_header HOST $host;
    proxy_set_header X-Forwarded-Proto $scheme;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

    proxy_pass http://example.com/new/prefix;
}

. . .
```

The above request sets the "Host" header to the `$host` variable, which should contain information about the original host being requested. The `X-Forwarded-Proto` header gives the proxied server information about the schema of the original client request (whether it was an http or an https request).

The `X-Real-IP` is set to the IP address of the client so that the proxy can correctly make decisions or log based on this information. The `X-Forwarded-For` header is a list containing the IP addresses of every server the client has been proxied through up to this point. In the example above, we set this to the `$proxy_add_x_forwarded_for` variable. This variable takes the value of the original `X-Forwarded-For` header retrieved from the client and adds the Nginx server's IP address to the end.

Of course, we could move the `proxy_set_header` directives out to the server or http context, allowing it to be referenced in more than one location:

```
# server context

proxy_set_header HOST $host;
proxy_set_header X-Forwarded-Proto $scheme;
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

location /match/here {
    proxy_pass http://example.com/new/prefix;
}

location /different/match {
    proxy_pass http://example.com;
```

```
}
```

Defining an Upstream Context for Load Balancing Proxied Connections

In the previous examples, we demonstrated how to do a simple http proxy to a single backend server. Nginx allows us to easily scale this configuration out by specifying entire pools of backend servers that we can pass requests to.

We can do this by using the `upstream` directive to define a pool of servers. This configuration assumes that any one of the listed servers is capable of handling a client's request. This allows us to scale out our infrastructure with almost no effort. The `upstream` directive must be set in the http context of your Nginx configuration.

Let's look at a simple example:

```
# http context

upstream backend_hosts {
    server host1.example.com;
    server host2.example.com;
    server host3.example.com;
}

server {
    listen 80;
    server_name example.com;

    location /proxy-me {
        proxy_pass http://backend_hosts;
    }
}
```

In the above example, we've set up an upstream context called `backend_hosts`. Once defined, this name will be available for use within proxy passes as if it were a regular domain name. As you can see, within our server block we pass any request made to `example.com/proxy-me/...` to the pool we defined above. Within that pool, a host is selected by applying a configurable algorithm. By default, this is just a simple round-robin selection process (each request will be routed to a different host in turn).

Changing the Upstream Balancing Algorithm

You can modify the balancing algorithm used by the upstream pool by including directives or flags within the upstream context:

- **(round robin)**: The default load balancing algorithm that is used if no other balancing directives are present. Each server defined in the upstream context is passed requests sequentially in turn.
- `least_conn`: Specifies that new connections should always be given to the backend that has the least number of active connections. This can be especially useful in situations where connections to the backend may persist for some time.
- `ip_hash`: This balancing algorithm distributes requests to different servers based on the client's IP address. The first three octets are used as a key to decide on the server to handle the request. The result is that clients tend to be served by the same server each time, which can assist in session consistency.
- `hash`: This balancing algorithm is mainly used with memcached proxying. The servers are divided based on the value of an arbitrarily provided hash key. This can be text, variables, or a combination. This is the only balancing method that requires the user to provide data, which is the key that should be used for the hash.

When changing the balancing algorithm, the block may look something like this:

```
# http context

upstream backend_hosts {

    least_conn;

    server host1.example.com;
    server host2.example.com;
    server host3.example.com;
}

. . .
```

In the above example, the server will be selected based on which one has the least connections. The `ip_hash` directive could be set in the same way to get a certain amount of session "stickiness".

As for the `hash` method, you must provide the key to hash against. This can be whatever you wish:

```
# http context

upstream backend_hosts {

    hash $remote_addr$remote_port consistent;

    server host1.example.com;
    server host2.example.com;
    server host3.example.com;
}

. . .
```

The above example will distribute requests based on the value of the client ip address and port. We also added the optional parameter `consistent`, which implements the ketama consistent hashing algorithm. Basically, this means that if your upstream servers change, there will be minimal impact on your cache.

Setting Server Weight for Balancing

In declarations of the backend servers, by default, each servers is equally "weighted". This assumes that each server can and should handle the same amount of load (taking into account the effects of the balancing algorithms). However, you can also set an alternative weight to servers during the declaration:

```
# http context

upstream backend_hosts {
    server host1.example.com weight=3;
    server host2.example.com;
    server host3.example.com;
}

. . .
```

In the above example, `host1.example.com` will receive three times the traffic as the other two servers. By default, each server is assigned a weight of one.

Using Buffers to Free Up Backend Servers

One issue with proxying that concerns many users is the performance impact of adding an additional server to the process. In most cases, this can be largely mitigated by taking advantage of Nginx's buffering and caching capabilities.

When proxying to another server, the speed of two different connections will affect the client's experience:

- The connection from the client to the Nginx proxy.
- The connection from the Nginx proxy to the backend server.

Nginx has the ability to adjust its behavior based on whichever one of these connections you wish to optimize.

Without buffers, data is sent from the proxied server and immediately begins to be transmitted to the client. If the clients are assumed to be fast, buffering can be turned off in order to get the data to the client as soon as possible. With buffers, the Nginx proxy will temporarily store the backend's response and then feed this data to the client. If the client is slow, this allows the Nginx server to close the connection to the backend sooner. It can then handle distributing the data to the client at whatever pace is possible.

Nginx defaults to a buffering design since clients tend to have vastly different connection speeds. We can adjust the buffering behavior with the following directives. These can be set in the http, server, or location contexts. It is important to keep in mind that the sizing directives are configured *per request*, so increasing them beyond your need can affect your performance when there are many client requests:

- **`proxy_buffering`**: This directive controls whether buffering for this context and child contexts is enabled. By default, this is "on".
- **`proxy_buffers`**: This directive controls the number (first argument) and size (second argument) of buffers for proxied responses. The default is to configure 8 buffers of a size equal to one memory page (either `4k` or `8k`). Increasing the number of buffers can allow you to buffer more information.
- **`proxy_buffer_size`**: The initial portion of the response from a backend server, which contains headers, is buffered separately from the rest of the response. This directive sets the size of the buffer for this portion of the response. By default, this will be the same size as `proxy_buffers`, but since this is used for header information, this can usually be set to a lower value.
- **`proxy_busy_buffers_size`**: This directive sets the maximum size of buffers that can be marked "client-ready" and thus busy. While a client can only read the data from one buffer at a time, buffers are placed in a queue to send to the client in bunches. This directive controls the size of the buffer space allowed to be in this state.
- **`proxy_max_temp_file_size`**: This is the maximum size, per request, for a temporary file on disk. These are created when the upstream response is too large to fit into a buffer.
- **`proxy_temp_file_write_size`**: This is the amount of data Nginx will write to the temporary file at one time when the proxied server's response is too large for the configured buffers.
- **`proxy_temp_path`**: This is the path to the area on disk where Nginx should store any temporary files when the response from the upstream server cannot fit into the configured buffers.

As you can see, Nginx provides quite a few different directives to tweak the buffering behavior. Most of the time, you will not have to worry about the majority of these, but it can be useful to adjust some of these values. Probably the most useful to adjust are the `proxy_buffers` and `proxy_buffer_size` directives.

An example that increases the number of available proxy buffers for each upstream request, while trimming down the buffer that likely stores the headers would look like this:

```
# server context
```

```
proxy_buffering on;
proxy_buffer_size 1k;
proxy_buffers 24 4k;
proxy_busy_buffers_size 8k;
proxy_max_temp_file_size 2048m;
proxy_temp_file_write_size 32k;

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

In contrast, if you have fast clients that you want to immediately serve data to, you can turn buffering off completely. Nginx will actually still use buffers if the upstream is faster than the client, but it will immediately try to flush data to the client instead of waiting for the buffer to pool. If the client is slow, this can cause the upstream connection to remain open until the client can catch up. When buffering is "off" only the buffer defined by the `proxy_buffer_size` directive will be used:

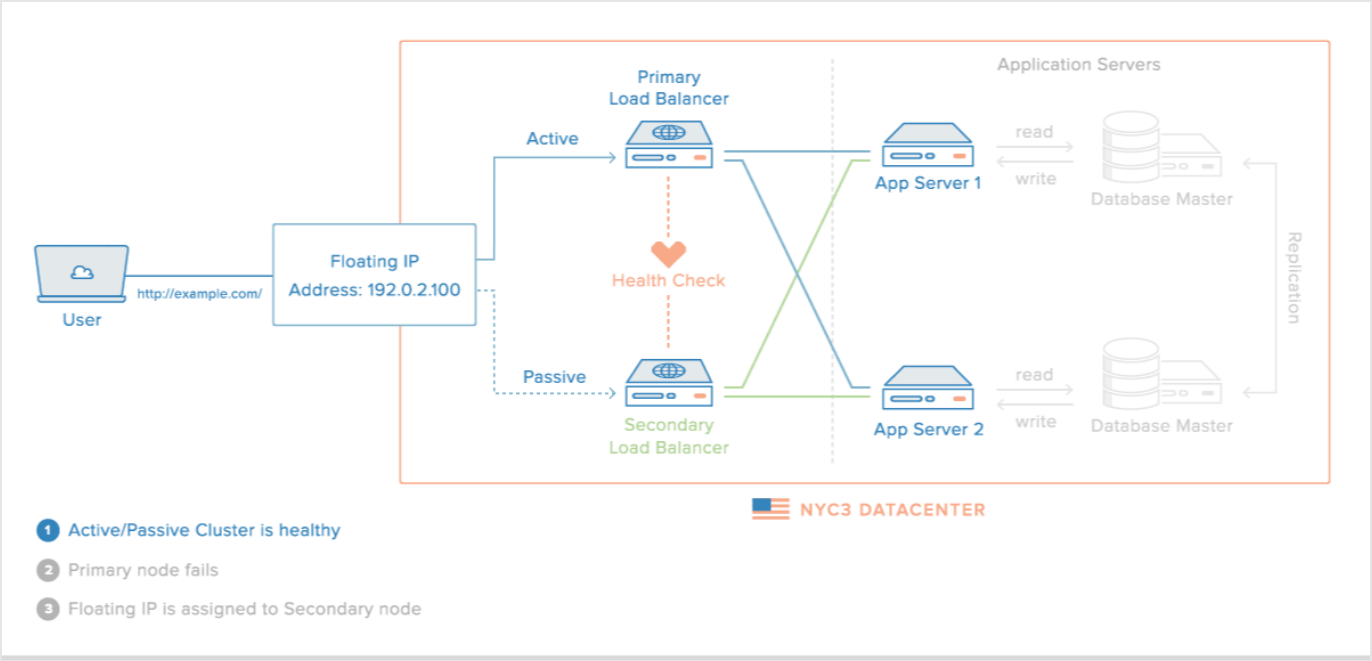
```
# server context

proxy_buffering off;
proxy_buffer_size 4k;

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

High Availability (Optional)

Nginx proxying can be made more robust by adding in a redundant set of load balancers, creating a high availability infrastructure.

A *high availability* (HA) setup is an infrastructure without a single point of failure, and your load balancers are a part of this configuration. By having more than one load balancer, you prevent potential downtime if your load balancer is unavailable or if you need to take them down for maintenance.

Here is a diagram of a basic high availability setup:



In this example, you have multiple load balancers (one active and one or more passive) behind a static IP address that can be remapped from one server to another. Client requests are routed from the static IP to the active load balancer, then on to your backend servers. To learn more, read this section of How To Use Floating IPs.

Configuring Proxy Caching to Decrease Response Times

While buffering can help free up the backend server to handle more requests, Nginx also provides a way to cache content from backend servers, eliminating the need to connect to the upstream at all for many requests.

Configuring a Proxy Cache

To set up a cache to use for proxied content, we can use the `proxy_cache_path` directive. This will create an area where data returned from the proxied servers can be kept. The `proxy_cache_path` directive must be set in the http context.

In the example below, we will configure this and some related directives to set up our caching system.

```
# http context

proxy_cache_path /var/lib/nginx/cache levels=1:2 keys_zone=backcache:8m max_size=50m;
proxy_cache_key "$scheme$request_method$host$request_uri$is_args$args";
proxy_cache_valid 200 302 10m;
proxy_cache_valid 404 1m;
```

With the `proxy_cache_path` directive, we have have defined a directory on the filesystem where we would like to store our cache. In this example, we've chosen the `/var/lib/nginx/cache` directory. If this directory does not exist, you can create it with the correct permission and ownership by typing:

```
sudo mkdir -p /var/lib/nginx/cache
sudo chown www-data /var/lib/nginx/cache
sudo chmod 700 /var/lib/nginx/cache
```

The `levels=` parameter specifies how the cache will be organized. Nginx will create a cache key by hashing the value of a key (configured below). The levels we selected above dictate that a single character directory (this will be the last character of the hashed value) with a two character subdirectory (taken from the next two characters from the end of the hashed value) will be created. You usually won't have to be concerned with the specifics of this, but it helps Nginx quickly find the relevant values.

The `keys_zone=` parameter defines the name for this cache zone, which we have called `backcache`. This is also where we define how much metadata to store. In this case, we are storing 8 MB of keys. For each megabyte, Nginx can store around 8000 entries. The `max_size` parameter sets the maximum size of the actual cached data.

Another directive we use above is `proxy_cache_key`. This is used to set the key that will be used to store cached values. This same key is used to check whether a request can be served from the cache. We are setting this to a combination of the scheme (http or https), the HTTP request method, as well as the requested host and URI.

The `proxy_cache_valid` directive can be specified multiple times. It allows us to configure how long to store values depending on the status code. In our example, we store successes and redirects for 10 minutes, and expire the cache for 404 responses every minute.

Now, we have configured the cache zone, but we still need to tell Nginx when to use the cache.

In locations where we proxy to a backend, we can configure the use of this cache:

```
# server context

location /proxy-me {
    proxy_cache backcache;
    proxy_cache_bypass $http_cache_control;
    add_header X-Proxy-Cache $upstream_cache_status;

    proxy_pass http://backend;
}

. . .
```

Using the `proxy_cache` directive, we can specify that the `backcache` cache zone should be used for this context. Nginx will check here for a valid entry before passing to the backend.

The `proxy_cache_bypass` directive is set to the `$http_cache_control` variable. This will contain an indicator as to whether the client is explicitly requesting a fresh, non-cached version of the resource. Setting this directive allows Nginx to correctly handle these types of client requests. No further configuration is required.

We also added an extra header called `X-Proxy-Cache`. We set this header to the value of the `$upstream_cache_status` variable. Basically, this sets a header that allows us to see if the request resulted in a cache hit, a cache miss, or if the cache was explicitly bypassed. This is especially valuable for debugging, but is also useful information for the client.

Notes about Caching Results

Caching can improve the performance of your proxy enormously. However, there are definitely considerations to keep in mind when configuring cache.

First, any user-related data should *not* be cached. This could result in one user's data being presented to another user. If your site is completely static, this is probably not an issue.

If your site has some dynamic elements, you will have to account for this in the backend servers. How you handle this depends on what application or server is handling the backend processing. For private content, you should set the `Cache-Control` header to "no-cache", "no-store", or "private" depending on the nature of the data:

- **no-cache**: Indicates that the response shouldn't be served again without first checking that the data hasn't changed on the backend. This can be used if the data is dynamic and important. An ETag hashed metadata header is checked on each request and the previous value can be served if the backend returns the same hash value.
- **no-store**: Indicates that at no point should the data received ever be cached. This is the safest option for private data, as it means that the data must be retrieved from the server every time.
- **private**: This indicates that no shared cache space should cache this data. This can be useful for indicating that a user's browser can cache the data, but the proxy server shouldn't consider this data valid for subsequent requests.
- **public**: This indicates that the response is public data that can be cached at any point in the connection.

A related header that can control this behavior is the `max-age` header, which indicates the number of seconds that any resource should be cached.

Setting these headers correctly, depending on the sensitivity of the content, will help you take advantage of cache while keeping your private data safe and your dynamic data fresh.

If your backend also uses Nginx, you can set some of this using the `expires` directive, which will set the `max-age` for `Cache-Control`:

```
location / {
    expires 60m;
}

location /check-me {
    expires -1;
}
```

In the above example, the first block allows content to be cached for an hour. The second block sets the `Cache-Control` header to “no-cache”. To set other values, you can use the `add_header` directive, like this:

```
location /private {
    expires -1;
    add_header Cache-Control "no-store";
}
```

Conclusion

Nginx is first and foremost a reverse proxy, which also happens to have the ability to work as a web server. Because of this design decision, proxying requests to other servers is fairly straight forward. Nginx is very flexible though, allowing for more complex control over your proxying configuration if desired.

By Justin Ellingwood

Related

TUTORIAL

How To Set Up Flask with MongoDB and Docker

As a micro web framework built on Python, Flask provides an extensible way

for developers to grow their applications through extensions that can be integrated into projects. To continue the scalability of a developer's tech stack, MongoDB is designed to scale and work with frequent changes. Developers can use Docker to simplify the process of packaging and developing their applications.

TUTORIAL

How To Build and Deploy Packages for Your FreeBSD Servers Using Buildbot and Poudriere

Poudriere is the standard tool on FreeBSD to build, test, and audit packages as well as maintain the package repositories. In this tutorial you'll configure Poudriere, build a set of FreeBSD packages, set up hosting, and automate the build using Buildbot as a continuous integration platform. Next you will access the packages from a client machine. Finally, if you want to integrate your own software builds with Poudriere, you'll set up a sample Nginx deployment on FreeBSD.

TUTORIAL

How To Autoscale Your Workloads on DigitalOcean Kubernetes

With Kubernetes applications that often have unexpected loads, the most efficient and least error-prone approach is to automate your clusters' scaling with the Horizontal Pod Autoscaler. In this tutorial, you will deploy a sample Nginx application to DigitalOcean Managed Kubernetes that can autoscale horizontally to account for increased CPU load. You will accomplish this by deploying Metrics Server to your cluster to gather pod metrics for HPA to use when deploying and

TUTORIAL

How To Deploy a PHP Application with Kubernetes on Ubuntu 18.04

Kubernetes is a open-source container orchestration system. It allows you to create, update, and scale containers without worrying about downtime. In this tutorial you will deploy a PHP 7 application with Nginx and PHP-FPM running in separate containers.

Still looking for an answer?

Ask a question Search for more help

42 Comments

Leave a comment...

Log In to Comment

chrishaas *November 26, 2014*

7 Very, very, very well written and thorough!

IvanMaglica *December 3, 2014*

0 Little correction: "key *zone" must be corrected to "keys* zone". Nginx does not very much information in errors.log

> IvanMaglica *December 3, 2014*
>
> 0 with underscores between keys and zone
>
> jellingwood *December 3, 2014*
>
> 0 Ah, nice catch. I've updated the article. Thanks @IvanMaglica!

jonathanwexler *December 21, 2014*

0 Hi Justin - really great tutorial.

Actually, what I am specifically trying to get running well is *NGINX server blocks using Docker containers* - with NGINX as both the host reverse proxy, and the web server inside the Docker containers - which are in turn all running Wordpress.

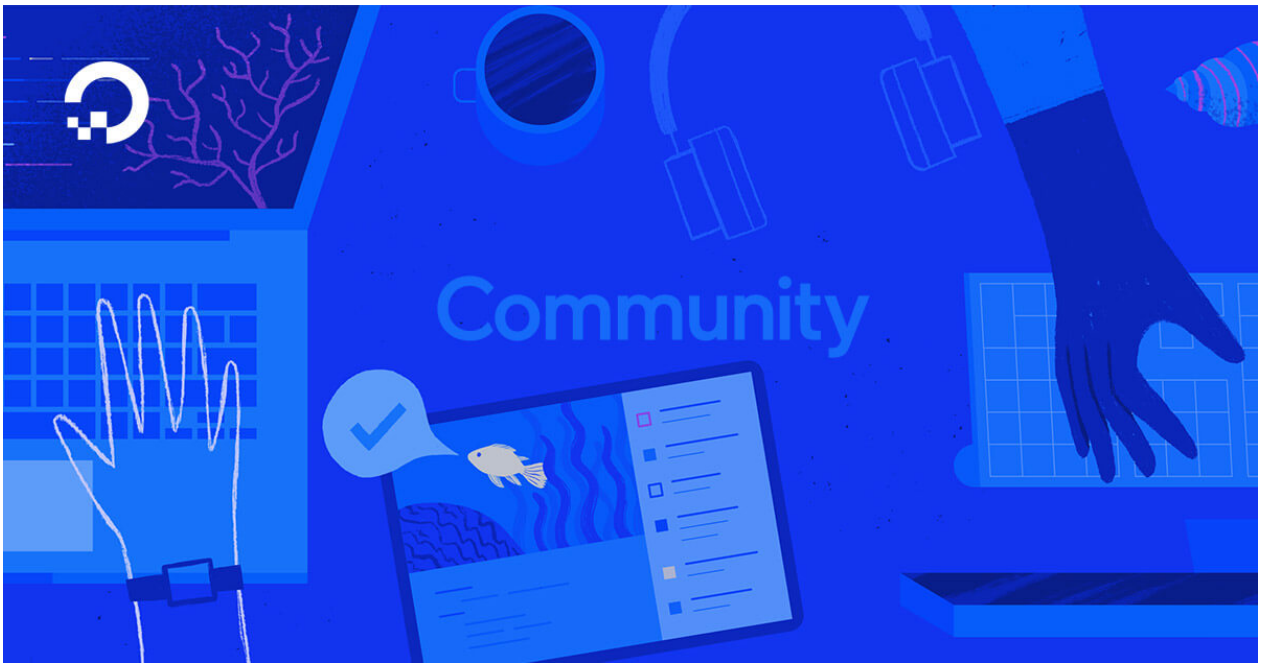
It would be great for me - and I suspect a lot of Digital Ocean users - if you could write such a tutorial - so basically *we co run many Wordpress Docker containers inside a single Droplet*.

If this can add some good luck - my best friend is also a technical writer named Justin - though he is from Montreal. Thanks profusely.

jellingwood *December 22, 2014*

1 @jonathanwexler: Although we don't have a guide for that configuration specifically, we do have one that covers how to use Nginx as a containerized reverse proxy.

Take a look through that guide and let us know what parts of the setup you want to try still need explanation. That would help us try to narrow the scope for a potential article.



Docker Explained: How To Containerize and Use Nginx as a Proxy

by O.S Tezer

In this DigitalOcean article, our goal is to learn about creating a docker container from a base image and building it to run Nginx (layer by layer). Later, following the steps from the beginning, we create a Dockerfile to automate the process using a custom input file for configurations. In the end, using this Nginx docker image, it becomes possible

jonathanwexler *December 22, 2014*

0 Thanks @jellingwood — I am working on notes to send you (I too am a technical writer!) - can you tell me where it is best to post them?

jellingwood *December 23, 2014*

0 @jonathanwexler: Oh, that's awesome! We have a page for suggestions here. We have a bit of a backlog at the moment, but that's the best way to get your idea onto our list of topics to cover.

Thanks again for the suggestions!

jonathanwexler *January 11, 2015*

0 FYI @jellingwood I sort of figured this all out myself.

I will try to compose an article on wordpress-docker containers-and DNS soon.

I am figuring out the last kinks, then would love to share it with the DigitalOcean community :)

jellingwood *January 11, 2015*

0 @jonathanwexler: Awesome! I'm glad you got that sorted. A+!

If you plan on having it on DigitalOcean, make sure you go through the proper channels *before* you begin writing.

This page will explain more of the details. Thanks again for the heads up and good luck!

nadavkav *January 11, 2015*

0 Thank you! illuminating and easy to read :-)

jcyrss *January 27, 2015*

0 Very good article, Justin. Thank you, that's what I've been looking for.

And I have another question about buffering.

Does Nginx also buffer http request from client? I mean, if Nginx get a http request from client, does it create connection with upstream server immediatly? Or it will create connection after it get a whole http request?

You know, many application server frameworks use worker-thread-pool model. If the client machine is far from Nginx and upstream server, without buffering http request in Nginx side, upstream server will waste pretty much CPU resource on waiting for completing a http request.

Thanks.

jellingwood *January 28, 2015*

0 @jcyrss: Nginx will buffer the data until the entire POST is received.

It keeps this data in a buffer (in RAM), which can be configured through the `client_body_buffer_size` directive. If the client request exceeds the value of this directive, it will be written to a temporary file until the entire file is received. Afterwards, i will relay the entire request to the upstream server.

I hope that helps.

pravinlogicap *March 4, 2015*

0 Awesome, It's very useful for me to use with NODE.JS also for the new learner, Fantastic :)

myemailum14 *March 26, 2015*

0 Hi, i've been trying to find this for hours. How do i get the server name chosen in upstream group at request. Thanks. $upstream_addr returns ip address

jellingwood *March 26, 2015*

0 @myemailum14: I'm not sure if I'm parsing your question correctly, but I think you may be looking for some of the variables available with the `upstream` module here. If not, can you give some more detail about exact information you are looking for?

myemailum14 *March 26, 2015*

0 @jellingwood hi this is what i want to do

say i have

upstream myapp1 {
server www.abc.com:80;
server www.asd.com:80;
}

```
server {
    listen 80;
    location / {
        proxy_set_header Host            XXxXXX;  //this
```

..
}

i want to set XXxxXXX as www.abc.com or www.asd.com based on whatever upstream server is chosen. I've been trying using conditonals, map, etc. But i'm new so im not getting anywhere.

the var upstreamd_addr gives ip not the domain name.

thanks for helping me out..

jellingwood *March 26, 2015*

0 @myemailum14: I'm having trouble thinking of a situation where you'd want to be doing domain-sensitive things during the load balancing process. Typically, if you have separate domains, you don't want to be using an `upstream` block to pass traffic between them. Instead, you'd want to set up a separate `server` block for each domain name and leave out the `upstream` block.

Basically, the issue that you're having is that `proxy_set_header` sets a header to pass to the upstream server *before* the upstream server is selected. So you're not going to be able to modify the `Host` header based on the chosen upstream server because one hasn't been selected yet.

At least from what you've described, it sounds like you'd be better off with something simple like this:

```
server {
    listen 80;
    server_name www.abc.com abc.com;

    . . .

}

server {
    listen 80;
    server_name www.asd.com asd.com;

    . . .

}
```

This allows you to serve two different sites with distinct content (or with the same content, if that's desired).

The `upstream` directive is more useful if 1) you have one domain name where all of your content will be requested and 2) you have multiple backend servers that each serve the same content, in order to spread out the load.

Hopefully I haven't misunderstood the problem.

myemailum14 *March 27, 2015*

0 if server hasn't been selected then how come i can get $upstream *addr right. I just wish i could get $upstream* url or something.

What i want to do is do load balancing AND use different sites. It's weird. But that's what i'm trying to do.

for reasons beyond my control however both asd.com and abc.com only respond if host header has their url in them.

therefore using map, conditional, variable, directive, default or whatever goes. I'm trying to set host header to the same domain. I can get upstream addr but not upstream url.

is there a way to program conditions in here or define default headers for each upstream.

jellingwood *March 27, 2015*

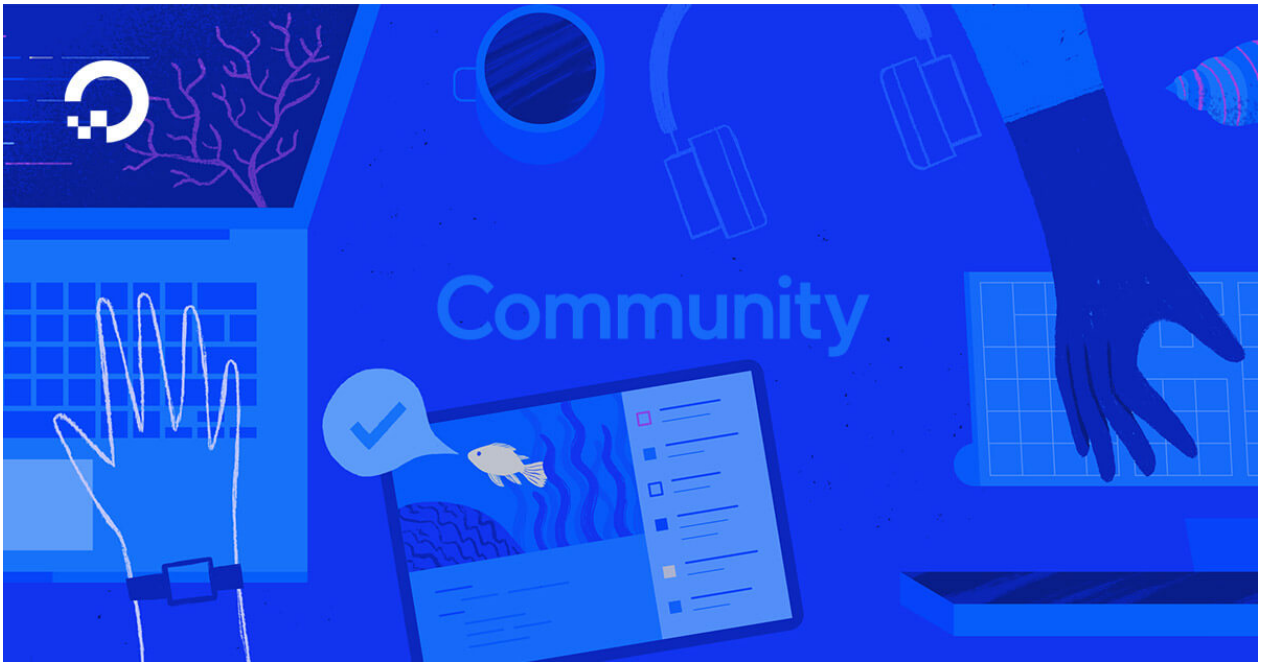
0 @myemailum14: I think this link is your best bet at getting something like that working. I don't really have any ideas that go beyond that. Maxim Dounin is an Nginx contributor and is incredibly knowledgeable about Nginx, so I would follow his advice closely.

jptoinf *April 30, 2015*

0 Is nginx free to use? And can we drop other caching like redis or varnish relying on nginx caching? And which is better Apache or nginx for handling high load?

newhosting2015 *June 30, 2015*

0 *Is nginx free to use?* ... Yes, it's a free open source web server.
And can we drop other caching like redis or varnish relying on nginx caching? Yes. Have a read of this tutorial for fastcgi caching. https://www.digitalocean.com/community/tutorials/how-to-setup-fastcgi-caching-with-nginx-on-your-vps
And which is better Apache or nginx for handling high load? It's a largely debated matter of opinion and of course depends on the multitude of configurations, but IMO Nginx rules the roost! Hope that helps and good luck.



How to Setup FastCGI Caching with Nginx on your VPS
by Jesin A

Here's how to setup FastCGI caching with Nginx on your VPS.

hoangdoan *June 10, 2015*

0 Thanks for your great article!

rhaynel *July 6, 2015*

0 Thank for the article, really help me!

adamachibo *July 9, 2015*

0 so sorry. I wanted to ask. whether this digitalocean service could breach the restriction speed my internet data plan after my internet data plan exceeded limit.?

simerng *August 8, 2015*

0 @jellingwood this is an awesome very well written article, probably the best available resource.

1. U mention nginx caching can replace redis caching. But since we shouldn't cache user data, wouldn't it be more beneficial to cache database results with redis and the sort as opposed to caching full url responses?
2. Can we define cache control in global/http context and then specify within location context a different value as an override?
3. How do the values specified in the proxy *cache*valid section affect the cache control directive?
4. Does the cache control header directive apply to just the server, or does the browser use this information for its own caching decisions?

moefar *August 15, 2015*

0 Well done!

mgvarley *August 25, 2015*

0 Thanks for the excellent article. I am building a SaaS offering and will have different types of client - some dedicated and some multi-tenant. I am considering using NGINX as an API Gateway and would like to isolate my dedicated clients on different servers / containers to my multi-tenant customers. Does NGINX allow you to route to different servers by API key?

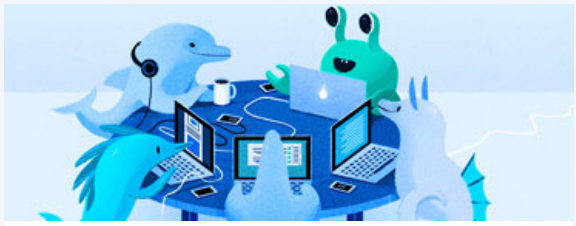
KevinYu *September 16, 2015*

0 Great tutorial.
I really appreciate it!

Load Mo Comments

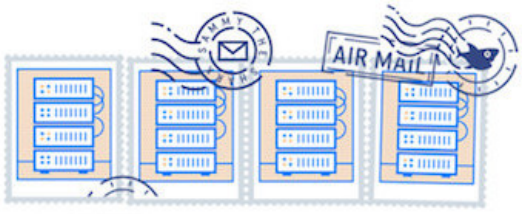
BECOME A CONTRIBUTOR

You get paid; we donate to tech nonprofits.



CONNECT WITH OTHER DEVELOPERS

Find a DigitalOcean Meetup near you.



GET OUR BIWEEKLY NEWSLETTER

Sign up for Infrastructure as a Newsletter.

Featured on Community

- Intro to Kubernetes
- Learn Python 3
- Machine Learning in Python
- Getting started with Go
- Migrate Node.js to Kubernetes

DigitalOcean Products

- Droplets
- Managed Databases
- Managed Kubernetes
- Spaces Object Storage
- Marketplace

Welcome to the developer cloud

DigitalOcean makes it simple to launch in the cloud and scale up as you grow – whether you're

running one virtual machine or ten thousand.

Learn More