Understanding the Nginx Configuration File Structure and Configuration Contexts

Posted November 19, 2014 573.2k NGINX CONCEPTUAL

By Justin Ellingwood
Become an author

Introduction

Nginx is a high performance web server that is responsible for handling the load of some of the largest sites on the internet. It is especially good at handling many concurrent connections and excels at serving static content.

While many users are aware of Nginx's capabilities, new users are often confused by some of the conventions they find in Nginx configuration files. In this guide, we will focus on discussing the basic structure of an Nginx configuration file along with some guidelines on how to design your files.

Understanding Nginx Configuration Contexts

This guide will cover the basic structure found in the main Nginx configuration file. The location of this file will vary depending on how you installed the software on your machine. For many distributions, the file will be located at `/etc/nginx/nginx.conf`. If it does not exist there, it may also be at `/usr/local/nginx/conf/nginx.conf` or `/usr/local/etc/nginx/nginx.conf`.

One of the first things that you should notice when looking at the main configuration file is that it appears to be

One of the first things that you should notice when looking at the main configuration file is that it appears to be organized in a tree-like structure, defined by sets of brackets (that look like `{` and `}`). In Nginx parlance, the areas that these brackets define are called "contexts" because they contain configuration details that are separated according to their area of concern. Basically, these divisions provide an organizational structure along with some conditional logic to decide whether to apply the configurations within.

Because contexts can be layered within one another, Nginx provides a level of directive inheritance. As a general rule, if a directive is valid in multiple nested scopes, a declaration in a broader context will be passed on to any child contexts as default values. The children contexts can override these values at will. It is worth noting that an override to any array-type directives will *replace* the previous value, not append to it.

Directives can only be used in the contexts that they were designed for. Nginx will error out on reading a configuration file with directives that are declared in the wrong context. The Nginx documentation contains information about which contexts each directive is valid in, so it is a great reference if you are unsure.

Below, we'll discuss the most common contexts that you're likely to come across when working with Nginx.

The Core Contexts

The first group of contexts that we will discuss are the core contexts that Nginx utilizes in order to create a hierarchical tree and separate the concerns of discrete configuration blocks. These are the contexts that comprise the major structure of an Nginx configuration.

The Main Context

The most general context is the "main" or "global" context. It is the only context that is not contained within the typical context blocks that look like this:

```
# The main context is here, outside any other contexts

. . .

context {

    . . .

}
```

Any directive that exist entirely outside of these blocks is said to inhabit the "main" context. Keep in mind that if your Nginx configuration is set up in a modular fashion, some files will contain instructions that appear to exist outside of a bracketed context, but which will be included within such a context when the configuration is stitched together.

The main context represents the broadest environment for Nginx configuration. It is used to configure details that affect the entire application on a basic level. While the directives in this section affect the lower contexts, many of these aren't *inherited* because they cannot be overridden in lower levels.

Some common details that are configured in the main context are the user and group to run the worker processes as, the number of workers, and the file to save the main process's PID. You can even define things like worker CPU affinity and the "niceness" of worker processes. The default error file for the entire application can be set at this level (this can be overridden in more specific contexts).

The Events Context

The "events" context is contained within the "main" context. It is used to set global options that affect how Nginx handles connections at a general level. There can only be a single events context defined within the Nginx

configuration.

This context will look like this in the configuration file, outside of any other bracketed contexts:

```
# main context

events {

    # events context
    . . .

}
```

Nginx uses an event-based connection processing model, so the directives defined within this context determine how worker processes should handle connections. Mainly, directives found here are used to either select the connection processing technique to use, or to modify the way these methods are implemented.

Usually, the connection processing method is automatically selected based on the most efficient choice that the platform has available. For Linux systems, the `epoll` method is usually the best choice.

Other items that can be configured are the number of connections each worker can handle, whether a worker will only take a single connection at a time or take all pending connections after being notified about a pending connection, and whether workers will take turns responding to events.

The HTTP Context

When configuring Nginx as a web server or reverse proxy, the “http” context will hold the majority of the configuration. This context will contain all of the directives and other contexts necessary to define how the program will handle HTTP or HTTPS connections.

The http context is a sibling of the events context, so they should be listed side-by-side, rather than nested. They both are children of the main context:

```
# main context

events {
    # events context

    . . .

}

http {
    # http context

    . . .

}
```

While lower contexts get more specific about how to handle requests, directives at this level control the defaults for every virtual server defined within. A large number of directives are configurable at this context and below, depending on how you would like the inheritance to function.

Some of the directives that you are likely to encounter control the default locations for access and error logs (`access_log` and `error_log`), configure asynchronous I/O for file operations (`aio`, `sendfile`, and `directio`), and configure the

server's statuses when errors occur (`error_page`). Other directives configure compression (`gzip` and `gzip_disable`), fine-tune the TCP keep alive settings (`keepalive_disable`, `keepalive_requests`, and `keepalive_timeout`), and the rules that Nginx will follow to try to optimize packets and system calls (`sendfile`, `tcp_nodelay`, and `tcp_nopush`). Additional directives configure an application-level document root and index files (`root` and `index`) and set up the various hash tables that are used to store different types of data (`*_hash_bucket_size` and `*_hash_max_size` for `server_names`, `types`, and `variables`).

The Server Context

The "server" context is declared *within* the "http" context. This is our first example of nested, bracketed contexts. It is also the first context that allows for multiple declarations.

The general format for server context may look something like this. Remember that these reside within the http context:

```
# main context

http {

    # http context

    server {

        # first server context

    }

    server {

        # second server context

    }

}
```

The reason for allowing multiple declarations of the server context is that each instance defines a specific virtual server to handle client requests. You can have as many server blocks as you need, each of which can handle a specific subset of connections.

Due to the possibility and likelihood of multiple server blocks, this context type is also the first that Nginx must use a selection algorithm to make decisions. Each client request will be handled according to the configuration defined in a single server context, so Nginx must decide which server context is most appropriate based on details of the request. The directives which decide if a server block will be used to answer a request are:

- **listen**: The ip address / port combination that this server block is designed to respond to. If a request is made by a client that matches these values, this block will potentially be selected to handle the connection.
- **server_name**: This directive is the other component used to select a server block for processing. If there are multiple server blocks with listen directives of the same specificity that can handle the request, Nginx will parse the "Host" header of the request and match it against this directive.

The directives in this context can override many of the directives that may be defined in the http context, including logging, the document root, compression, etc. In addition to the directives that are taken from the http context, we also can configure files to try to respond to requests (`try_files`), issue redirects and rewrites (`return` and `rewrite`), and set arbitrary variables (`set`).

The Location Context

The next context that you will deal with regularly is the location context. Location contexts share many relational qualities with server contexts. For example, multiple location contexts can be defined, each location is used to handle a certain type of client request, and each location is selected by virtue of matching the location definition against the client request through a selection algorithm.

While the directives that determine whether to select a server block are defined within the server *context*, the component that decides on a location's ability to handle a request is located in the location *definition* (the line that opens the location block).

The general syntax looks like this:

```
location match_modifier location_match {
    . . .
}
```

Location blocks live within server contexts and, unlike server blocks, can be nested inside one another. This can be useful for creating a more general location context to catch a certain subset of traffic, and then further processing it based on more specific criteria with additional contexts inside:

```
# main context
server {
    # server context
    location /match/criteria {
        # first location context
    }
    location /other/criteria {
        # second location context
        location nested_match {
            # first nested location
        }
        location other_nested {
            # second nested location
        }
    }
}
```

While server contexts are selected based on the requested IP address/port combination and the host name in the "Host" header, location blocks further divide up the request handling within a server block by looking at the request

URI. The request URI is the portion of the request that comes after the domain name or IP address/port combination.

So, if a client requests `http://www.example.com/blog` on port 80, the `http`, `www.example.com`, and port 80 would all be used to determine which server block to select. After a server is selected, the `/blog` portion (the request URI), would be evaluated against the defined locations to determine which further context should be used to respond to the request.

Many of the directives you are likely to see in a location context are also available at the parent levels. New directives at this level allow you to reach locations outside of the document root (`alias`), mark the location as only internally accessible (`internal`), and proxy to other servers or locations (using http, fastcgi, scgi, and uwsgi proxying).

Other Contexts

While the above examples represent the essential contexts that you will encounter with Nginx, other contexts exist as well. The contexts below were separated out either because they depend on more optional modules, they are used only in certain circumstances, or they are used for functionality that most people will not be using.

We will *not* be discussing each of the available contexts though. The following contexts will not be discussed in any depth:

- `split_clients`: This context is configured to split the clients that the server receives into categories by labeling them with variables based on a percentage. These can then be used to do A/B testing by providing different content to different hosts.
- `perl / perl_set`: These contexts configures Perl handlers for the location they appear in. This will only be used for processing with Perl.
- `map`: This context is used to set the value of a variable depending on the value of another variable. It provides a mapping of one variable's values to determine what the second variable should be set to.
- `geo`: Like the above context, this context is used to specify a mapping. However, this mapping is specifically used to categorize client IP addresses. It sets the value of a variable depending on the connecting IP address.
- `types`: This context is again used for mapping. This context is used to map MIME types to the file extensions that should be associated with them. This is usually provided with Nginx through a file that is sourced into the main `nginx.conf` config file.
- `charset_map`: This is another example of a mapping context. This context is used to map a conversion table from one character set to another. In the context header, both sets are listed and in the body, the mapping takes place.

The contexts below are not as common as the ones we have discussed so far, but are still very useful to know about.

The Upstream Context

The upstream context is used to define and configure "upstream" servers. Basically, this context defines a named pool of servers that Nginx can then proxy requests to. This context will likely be used when you are configuring proxies of various types.

The upstream context should be placed within the http context, outside of any specific server contexts. The general form looks something like this:

```
# main context

http {

    # http context

    upstream upstream_name {
```

```
    # upstream context

    server proxy_server1;
    server proxy_server2;

    . . .

  }

  server {

    # server context

  }

}
```

The upstream context can then be referenced by name within server or location blocks to pass requests of a certain type to the pool of servers that have been defined. The upstream will then use an algorithm (round-robin by default) to determine which specific server to hand the request to. This context gives our Nginx the ability to do some load balancing when proxying requests.

The Mail Context

Although Nginx is most often used as a web or reverse proxy server, it can also function as a high performance mail proxy server. The context that is used for directives of this type is called, appropriately, “mail”. The mail context is defined within the “main” or “global” context (outside of the http context).

The main function of the mail context is to provide an area for configuring a mail proxying solution on the server. Nginx has the ability to redirect authentication requests to an external authentication server. It can then provide access to POP3 and IMAP mail servers for serving the actual mail data. The mail context can also be configured to connect to an SMTP Relayhost if desired.

In general, a mail context will look something like this:

```
# main context

events {

  # events context

}

mail {

  # mail context

}
```

The If Context

The “if” context can be established to provide conditional processing of directives defined within. Like an if statement in conventional programming, the if directive in Nginx will execute the instructions contained if a given test returns “true”.

The if context in Nginx is provided by the rewrite module and this is the primary intended use of this context. Since

Nginx will test conditions of a request with many other purpose-made directives, if should **not** be used for most forms of conditional execution. This is such an important note that the Nginx community has created a page called if is evil.

The problem is basically that the Nginx processing order can very often lead to unexpected results that seem to subvert the meaning of an if block. The only directives that are considered reliably safe to use inside of these contexts are the `return` and `rewrite` directives (the ones this context was created for). Another thing to keep in mind when using an if context is that it renders a `try_files` directive in the same context useless.

Most often, an if will be used to determine whether a rewrite or return is needed. These will most often exist in location blocks, so the common form will look something like this:

```
# main context

http {

    # http context

    server {

        # server context

        location location_match {

            # location context

            if (test_condition) {

                # if context

            }

        }

    }

}
```

The Limit_except Context

The `limit_except` context is used to restrict the use of certain HTTP methods within a location context. For example, if only certain clients should have access to POST content, but everyone should have the ability to read content, you can use a `limit_except` block to define this requirement.

The above example would look something like this:

```
. . .

# server or location context

location /restricted-write {

    # location context

    limit_except GET HEAD {

        # limit_except context

        allow 192.168.1.1/24;
```

```
        allow 192.168.1.1/24;
        deny all;
    }
}
```

This will apply the directives inside the context (meant to restrict access) when encountering any HTTP methods **except** those listed in the context header. The result of the above example is that any client can use the GET and HEAD verbs, but only clients coming from the `192.168.1.1/24` subnet are allowed to use other methods.

General Rules to Follow Regarding Contexts

Now that you have an idea of the common contexts that you are likely to encounter when exploring Nginx configurations, we can discuss some best practices to use when dealing with Nginx contexts.

Apply Directives in the Highest Context Available

Many directives are valid in more than one context. For instance, there are quite a few directives that can be placed in the http, server, or location context. This gives us flexibility in setting these directives.

However, as a general rule, it is usually best to declare directives in the highest context to which they are applicable, and overriding them in lower contexts as necessary. This is possible because of the inheritance model that Nginx implements. There are many reasons to use this strategy.

First of all, declaring at a high level allows you to avoid unnecessary repetition between sibling contexts. For instance, in the example below, each of the locations is declaring the same document root:

```
http {
    server {
        location / {
            root /var/www/html;

            . . .

        }

        location /another {
            root /var/www/html;

            . . .

        }

    }
}
```

You could move the root out to the server block, or even to the http block, like this:

```
http {
    root /var/www/html;
    server {
        location / {

            . . .

        }

        location /another {
```

```
        ...

    }
  }
}
```

Most of the time, the server level will be most appropriate, but declaring at the higher level has its advantages. This not only allows you to set the directive in fewer places, it also allows you to cascade the default value down to all of the child elements, preventing situations where you run into an error by forgetting a directive at a lower level. This can be a major issue with long configurations. Declaring at higher levels provides you with a sane default.

Use Multiple Sibling Contexts Instead of If Logic for Processing

When you want to handle requests differently depending on some information that can be found in the client's request, often users jump to the "if" context to try to conditionalize processing. There are a few issues with this that we touched on briefly earlier.

The first is that the "if" directive often return results that do not align with the administrator's expectations. Although the processing will always lead to the same result given the same input, the way that Nginx interprets the environment can be vastly different than can be assumed without heavy testing.

The second reason for this is that there are already optimized, purpose-made directives that are used for many of these purposes. Nginx already engages in a well-documented selection algorithm for things like selecting server blocks and location blocks. So if it is possible, it is best to try to move your different configurations into their own blocks so that this algorithm can handle the selection process logic.

For instance, instead of relying on rewrites to get a user supplied request into the format that you would like to work with, you should try to set up two blocks for the request, one of which represents the desired method, and the other that catches messy requests and redirects (and possibly rewrites) them to your correct block.

The result is usually easier to read and also has the added benefit of being more performant. Correct requests undergo no additional processing and, in many cases, incorrect requests can get by with a redirect rather than a rewrite, which should execute with lower overhead.

Conclusion

By this point, you should have a good grasp on Nginx's most common contexts and the directive that create the blocks that define them.

Always check Nginx's documentation for information about which contexts a directive can be placed in and to evaluate the most effective location. Taking care when creating your configurations will not only increase maintainability, but will also often increase performance.

By Justin Ellingwood

Report an issue

Tutorial Series

Introduction to Nginx and LEMP on Ubuntu 14.04

This tutorial series helps sysadmins set up a new web server using the LEMP stack, focusing on Nginx setup with virtual blocks. This will let you serve multiple websites from one Droplet. You'll start by setting up your Ubuntu 14.04 server and end with multiple virtual blocks set up for your websites. An Nginx configuration guide is included at the end for reference.

Show Tutorials

Related

TUTORIAL

Understanding Nginx Server and Location Block Selection Algorithms

Nginx is one of the most popular web servers in the world. In this guide, we will discuss how Nginx selects the server and location block that will handle a given client's request. We will cover the algorithm in place, as well as the directives and options you can use to modify the selection process.

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

TUTORIAL

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

Developers can use Poudriere, a 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 desired packages, set up HTTP-base package 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 this process, you'll step through building an FreeBSD package with Nginx and PHP-FPM running in separate containers.

TUTORIAL

How To Deploy a PHP Application with Kubernetes on Ubuntu 18.04

Kubernetes is an 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 on a Kubernetes cluster with Nginx and PHP-FPM running in separate containers.

Still looking for an answer?

Ask a question | Search for more help

19 Comments

Leave a comment...

Log In to Comment

i524176 *February 19, 2015*

0 Thanks,i Love nginx.

peter.idah *March 26, 2015*

2 Nice article; minor point – Allowing the GET method makes the HEAD method also allowed: http://nginx.org/en/docs/http/ngx_http_core_module.html#limit_except

united *April 16, 2016*

0 Very basic question: if I place an nginx.conf in the root of my domain as such `/var/www/domain.com/nginx.conf` with contents like

```
location ~ ^/\.user\.ini {
    deny all;
}
```

will those contexts and directives be concatenated and executed with the main nginx.conf file located here: `/etc/nginx/nginx.conf`

Hope this is a clear question and thank you in advance...

kamaln7 MOD *April 16, 2016*

0 Hi @united,

Unlike Apache, nginx does not support .htaccess or similar files. The page I linked also gives an explanation for why it is not supported. So, unfortunately, you will have to edit the virtualhost config (`/etc/nginx/sites-enabled` if your nginx setup uses that, `/etc/nginx/nginx.conf` otherwise).

mimrahe *July 14, 2016*

0 Digital Ocean Saves the Life!

hedonplay *July 27, 2016*

0 You explain it really well!

skippyV *August 23, 2016*

0 I appreciate the well formulated descriptions on this page! However this sentence “While the directives that determine whether to select a server block are defined within the server context itself, the component that decides on a location’s ability to handle a request is located in the location definition itself.” makes me question it’s message. It is worded as a comparison that focuses on a difference: between directives of a server block and directives of a location block. But I think I’m missing it’s meaning. Can you reword it to emphasize the difference?

jellingwood *August 23, 2016*

0 @skippyV: Yep, I just updated it a bit. I agree that it was pretty unclear. Basically, I was just trying to communicate that the defining matching criteria for a server block comes from directives defined within the block. In contrast, the matching criteria for location blocks are defined in the header that opens the block.

skippyV *August 24, 2016*

0 Ah I see. The line that precedes the block vs statements within the block. Thanks for clarifying that!

pyvkd *October 12, 2016*

0 Great article @jellingwood .

A tiny edit though in the server context section first example you left a semicolon after http context.

jellingwood *October 12, 2016*

0 @pyvkd: Good catch! Thanks!

hudsantos *March 29, 2017*

0 link to page If Is Evil changed to: https://www.nginx.com/resources/wiki/start/topics/depth/ifisevil/

jellingwood *March 29, 2017*

0 @hudsantos Thanks for the heads up! I've updated the link in the article.

umbobabo77 *April 11, 2017*

0 Good article, thank you for sharing.
What is not clear to me is how Nginx behaves when there are multiple matching location directives that are siblings.
Example:
Let's suppose the URI is '/blog/article'

location ~ ^\/blog {
... do something ...
}

location ~ \/article {
... do something else ...
}

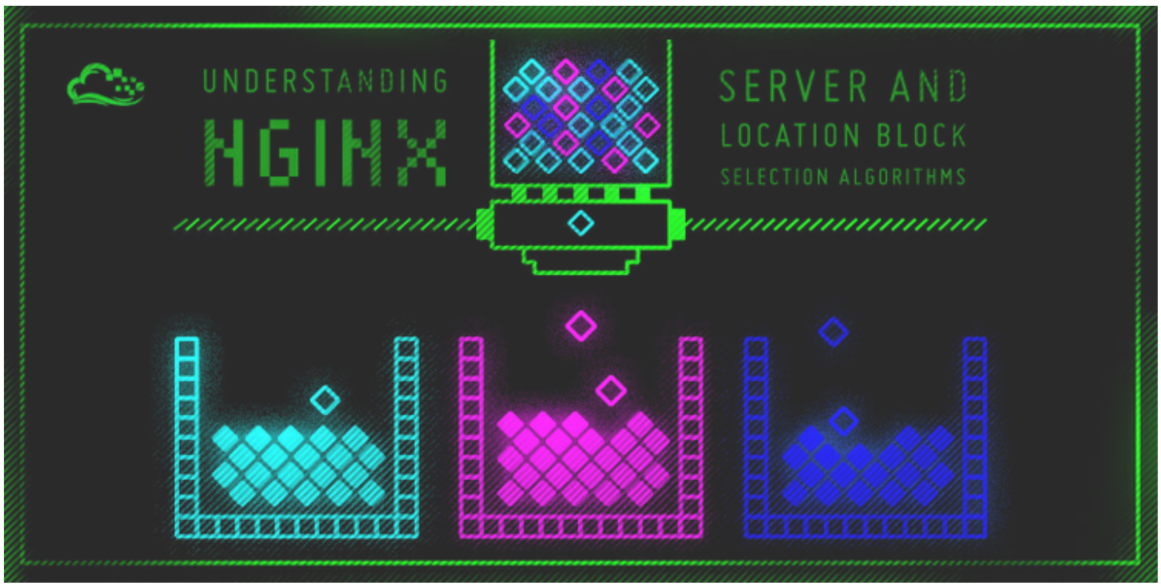
Could you explain to me please?

jellingwood *April 11, 2017*

0 @umbobabo77 Hey there. This article covers the way that Nginx selects location blocks in some depth if you want to take look.

In short though, in your specific example, you're using case-sensitive regular expression matching. Assuming that those are the only two location blocks in the server block context, the `location ~ ^\/blog` block would be selected. After finding the longest non-regular expression matches, Nginx looks for regular expression matches. If any are found, it will use the first matching block.

The linked to article explains this in more detail, but that's the basic idea. Hope that helps!



Understanding Nginx Server and Location Block Selection Algorithms
by Justin Ellingwood

Nginx is one of the most popular web servers in the world. In this guide, we will discuss how Nginx selects the serv and location block that will handle a given client's request. We will go over the algorithm in place, as well as the directives and options you can use to modify the selection process.

umbobabo77 *April 12, 2017*

0 Thank you very much. Really appreciated.

hooda *September 2, 2017*

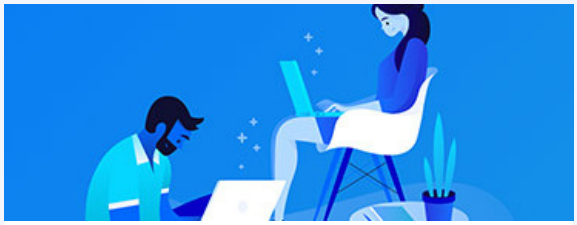
0 Very relevant article. Thanks.
I've added this to other nginx recommended tutorials at https://hackr.io/tutorials/learn-nginx

zx1986 *December 5, 2017*

0 Nice post!

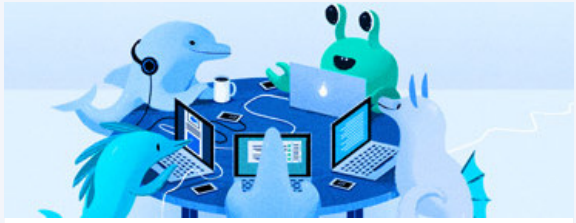
rochie94 *October 20, 2018*

0 thanks for this!!!



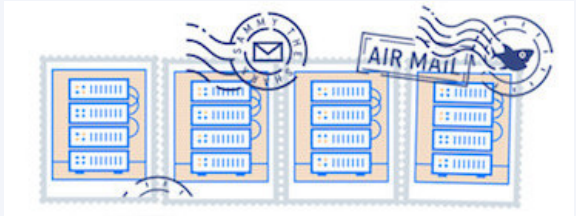
BECOME A CONTRIBUTOR

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

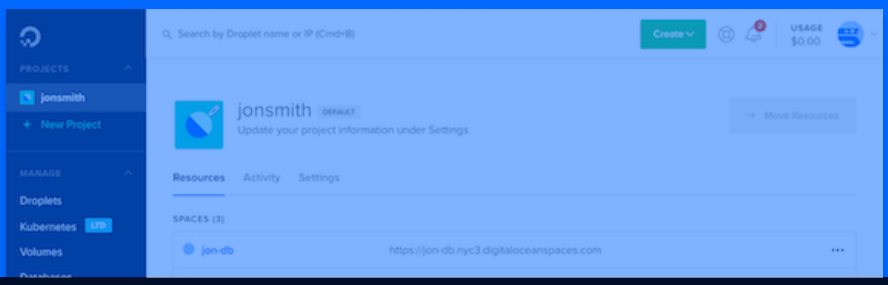
- Intro to Kubernetes
- Learn Python 3
- Machine Learning in Python
- Getting started with Go
- Migrate Node.js to Kubernetes

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



Company

About
Leadership
Blog
Careers
Partners
Referral Program
Press
Legal & Security

Products

Products Overview
Pricing
Droplets
Kubernetes
Managed Databases
Spaces
Marketplace
Load Balancers