How to Configure NGINX for a Flask Web Application

SEPTEMBER 26, 2016 / PATKENNEDY79@GMAIL.COM / 3 COMMENTS

Introduction

In this blog post, I'll be explaining what NGINX is and how to configure it for serving a Flask web application. This blog post is part of a larger series on deploying Flask applications. I've found a lot of documentation about NGINX and how to configure it, but I wanted to dive into the details for how NGINX can be used in a Flask web application and how to configure it. I've found the configuration of NGINX to be a bit confusing, as a lot of the documentation simply shows a configuration file(s) without explaining the details of what each step does. Hopefully this blog post provides some clarity on configuring NGINX for your application.

What is NGINX?

From the NGINX (pronounced 'engine-X') website, here is the high-level description of the tool:

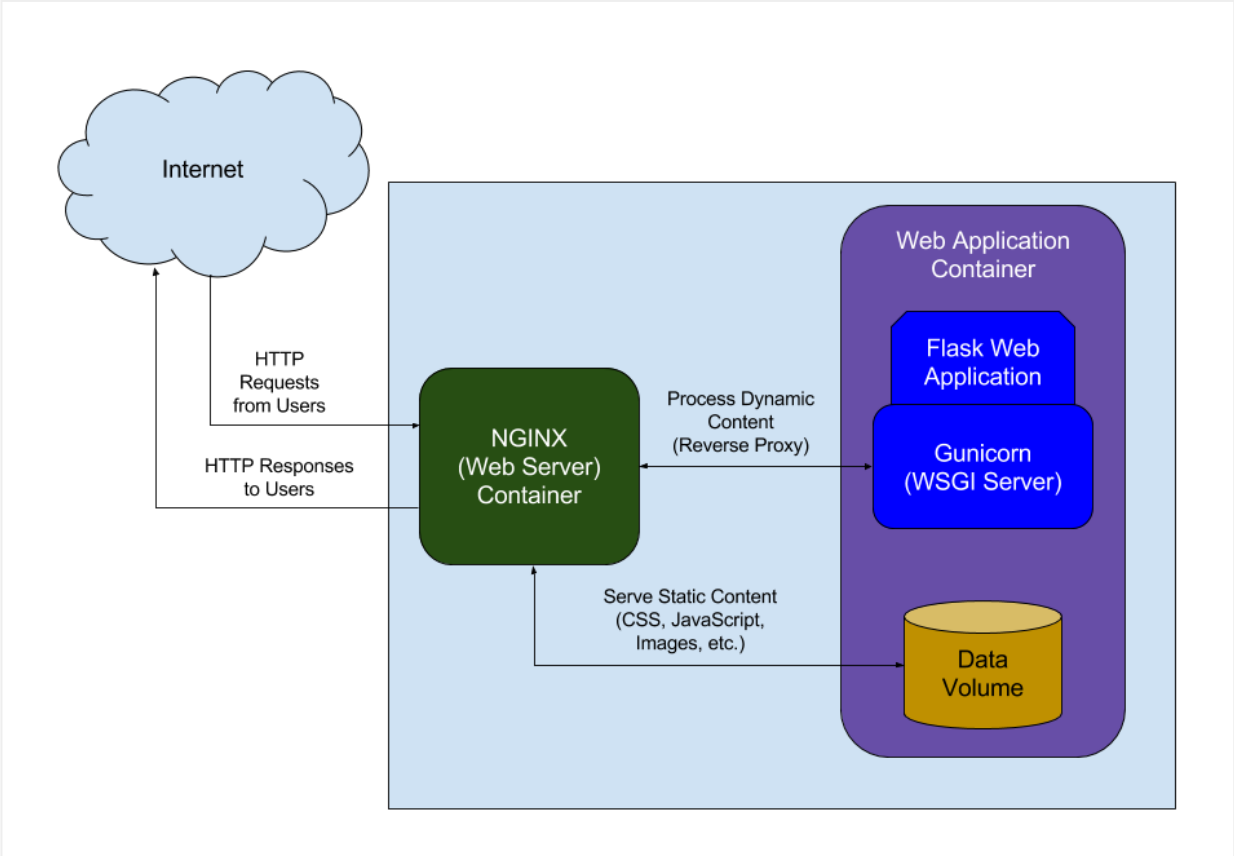
NGINX is a free, open-source, high-performance HTTP server and reverse proxy, as well as an IMAP/POP3 proxy server. NGINX is known for its high performance, stability, rich feature set, simple configuration, and low resource consumption.

Let's expand on this description… NGINX is a server that handles HTTP requests for your web application. For a typical web application, NGINX can be configured to perform the following with these HTTP requests:

- Reverse proxy the request to an upstream server (such as Gunicorn, uWsgi, Apache, etc.)
- Server static content (Javascript files, CSS files, images, documents, static HTML files)

NGINX also provides a load balancing capability to allow requests to be serviced by multiple upstream servers, but that functionality is not discussed in this blog post.

Here's a diagram illustrating how NGINX fits into a Flask web application:



NGINX handles the HTTP requests that come in from the internet (ie. the users of your application). Based on how you configure NGINX, it can directly provide the static content (Javascript files, CSS files, images, documents, static HTML files) back to the requester. Additionally, it can reverse proxy the requests to your WSGI (Web Server Gateway Interface) server to allow you to generate the dynamic content (HTML) in your Flask web application to be delivered back to the user.

This diagram assumes the use of Docker, but the configuration of NGINX would be very similar if not using Docker (just omit the concept of containers from the diagram).

Why do you need NGINX and Gunicorn?

NGINX is a HTTP server that is used in lots of different application stacks. It performs a lot of functions, but it is not able to directly interface with a Flask application. That is where Gunicorn comes in to play. HTTP requests are received by NGINX and passed along to Gunicorn to be processed by your Flask application (think of the route(s) defined in your views.py). Gunicorn is a WSGI server that handles HTTP requests and routes them to any python application that is WSGI-compliant, such as Flask, Django, Pyramid, etc.

Structure of NGINX Configuration Files

NOTE: This blog post uses NGINX v1.11.3. The configuration files could be located at different locations depending on your specific version on NGINX, such as /opt/nginx/conf/.

Depending on how you installed or are using NGINX, the structure of the configuration files will be slightly different. Both structures are presented below…

Structure 1

If you compile NGINX from the source code or use an official Docker image, then the configuration files are located at: /etc/nginx/ and the main configuration file is /etc/nginx/nginx.conf. At the bottom of /etc/nginx/nginx.conf is a line to include any additional configuration files located in the /etc/nginx/conf.d/ directory:

- include /etc/nginx/conf.d/*.conf;

Structure 2

If you installed NGINX using a package manager (such as apt-get on Ubuntu), then you will also have the following sub-directories in the /etc/nginx/ directory:

- sites-available – contains the different configuration files, often for different sites.

- sites-enabled – contains a symbolic link a file defined in sites-available

These directories are holdovers from Apache that have been applied to the configuration of NGINX.

Since the Flask applications that we're developing are using Docker, we'll be focusing on 'Structure 1' in this blog post.

NGINX Configuration

The top-level configuration file for NGINX is nginx.conf. NGINX allows for multiple layers of configuration files, which allows a lot of flexibility in configuring it just right for your application. For specific details about a parameter, the NGINX documentation provides a nice reference.

The configuration parameters for NGINX are grouped into blocks. Here are the blocks that we'll be working with in this blog post:

- Main – defined in nginx.conf (anything not defined in a block)
- Events – defined in nginx.conf
- Http – defined in nginx.conf
- Server – defined in _application_name_.conf

The breakdown of these blocks into different files allows you to define the high-level configuration parameters of NGINX in nginx.conf and the specific parameters for a virtual host(s)/server(s) to be in a *.conf file(s) that is specific to your web application.

Details of nginx.conf

The default version of nginx.conf that comes with the installation of NGINX is a good starting point for most servers. Let's investigate the details of nginx.conf and see how to expand upon the default settings…

Main Section

The main section (ie. configuration parameters not defined within blocks) of nginx.conf is:

```
user  nginx;
worker_processes  1;
```

```
error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;
```

The first parameter (user) defines the user that will own and run the Nginx server. This default value is good to use, especially when working with NGINX via a Docker container.

The second parameter (worker_processes) defines the number of worker processes. A recommended value for this parameter is the number of cores that are being used by your server. For a basic virtual private server (VPS), the default value of 1 is a good choice. Increment this number as you expand the performance of your VPS.

The third parameter (error_log) defines the location on the file system of the error log, plus a bonus parameter for the minimum severity to log messages for. The default value for this parameter is good.

The fourth parameter (pid) defines the file that will store the process ID of the main NGINX process. No need to change this default value.

events Block

The events block defines the parameters that affect connection processing. The events block is the first block in the nginx.conf file:

```
events {
    worker_connections  1024;
}
```

This block has a single parameter (worker_connections), which defines the maximum number of simultaneous connections that can be opened by a worker process. The default value for this parameter is good, as this defines 1024 total connections (but you have to count connections with users requesting sites and connections with the WSGI server).

http Block

The http block defines a number of parameters for how NGINX should handle HTTP web traffic. The http block is the second block in the nginx.conf file:

```
http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;
```

```
    log_format  main  '$remote_addr - $remote_user [$time_loca
                      '$status $body_bytes_sent "$http_referer
                      '"$http_user_agent" "$http_x_forwarded_f

    access_log  /var/log/nginx/access.log  main;

    sendfile        on;
    #tcp_nopush     on;

    keepalive_timeout  65;

    #gzip  on;

    include /etc/nginx/conf.d/*.conf;
}
```

The first parameter (include) specifies a configuration file to include, which is located at /etc/nginx/mime.types. This configuration files defines a long list of file types that are supported by NGINX. The default value should be kept for this parameter.

The second parameter (default_type) specifies the default file type that is returned to the user. For a Flask application that is generating dynamic HTML files, this parameter should be changed to: default_type text/html;

The third parameter (log_format) specifies the format of log messages. The default value should be kept for this parameter.

The fourth parameter (access_log) specifies the location of the log of access attempts to NGINX. The default value should be kept for this parameter.

The fifth parameter (send_file) and sixth parameter (tcp_nopush) start to get a bit more complicated. See this blog post about optimizing NGINX to get more details on these parameters (plus tcp_nodelay). Since we're planning to use NGINX to deliver static content, we should set these parameters as such:

```
sendfile        on;
tcp_nopush     on;
tcp_nodelay    on;
```

The seventh parameter (keepalive_timeout) defines the timeout value for keep-alive connections with the client. The default value should be kept for this parameter.

The eighth parameter (gzip) defines the usage of the gzip compression algorithm to reduce the amount of data to transmit. This reduction in data size is offset by an

increase in processing needed to perform the compression. The default value (off) should be kept for this parameter.

The ninth (and last) parameter (include) defines additional configuration files (ending in *.conf) from /etc/nginx/conf.d/. We'll now see how to use these additional configuration files to define the serving of static content and to define the reverse proxy to our WSGI server.

Final Configuration of nginx.conf

By taking the default version on nginx.conf and adjusting a few parameters for our needs (plus adding comments), here is the final version of nginx.conf:

```
# Define the user that will own and run the Nginx server
user  nginx;
# Define the number of worker processes; recommended value is
# cores that are being used by your server
worker_processes  1;

# Define the location on the file system of the error log, plu
# severity to log messages for
error_log  /var/log/nginx/error.log warn;
# Define the file that will store the process ID of the main N
pid        /var/run/nginx.pid;


# events block defines the parameters that affect connection p
events {
   # Define the maximum number of simultaneous connections tha
   worker_connections  1024;
}


# http block defines the parameters for how NGINX should handl
http {
   # Include the file defining the list of file types that are
   include       /etc/nginx/mime.types;
   # Define the default file type that is returned to the user
   default_type  text/html;

   # Define the format of log messages.
   log_format  main  '$remote_addr - $remote_user [$time_local
                     '$status $body_bytes_sent "$http_referer"
                     '"$http_user_agent" "$http_x_forwarded_fo

   # Define the location of the log of access attempts to NGIN
   access_log  /var/log/nginx/access.log  main;

   # Define the parameters to optimize the delivery of static
   sendfile        on;
   tcp_nopush     on;
   tcp_nodelay    on;
```

```
    # Define the timeout value for keep-alive connections with
    keepalive_timeout  65;

    # Define the usage of the gzip compression algorithm to red
    #gzip  on;

    # Include additional parameters for virtual host(s)/server(
    include /etc/nginx/conf.d/*.conf;
}
```

Configuring NGINX for Serving Static Content and as a Reverse Proxy

If you look at the default version of /etc/nginx/conf.g/default.conf, it defines the server block and provides a simple configuration with a lot of options to uncomment if you chose. Instead of going through each item in this file, let's discuss the key parameters that are needed for configuring NGINX to deliver static content and for reverse proxying the requests to our WSGI server. Here are the contents of _application_name_.conf that are recommended:

```
# Define the parameters for a specific virtual host/server
server {
    # Define the directory where the contents being requested a
    # root /usr/src/app/project/;

    # Define the default page that will be served If no page wa
    # (ie. if www.kennedyfamilyrecipes.com is requested)
    # index index.html;

    # Define the server name, IP address, and/or port of the se
    listen 80;
    # server_name xxx.yyy.zzz.aaa

    # Define the specified charset to the “Content-Type” respor
    charset utf-8;

    # Configure NGINX to deliver static content from the specif
    location /static {
        alias /usr/src/app/project/static;
    }

    # Configure NGINX to reverse proxy HTTP requests to the ups
    location / {
        # Define the location of the proxy server to send the r
        proxy_pass http://web:8000;

        # Redefine the header fields that NGINX sends to the up
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded

        # Define the maximum file size on file uploads
        client_max_body_size 5M;
```

```
    }
}
```

The server block defines the parameters for a specific virtual host/server, which is typically the single web application that you are hosting on a VPS.

The first parameter (root) defines the directory where the contents being requested are stored. NGINX will start looking in this directory when it receives a request from a user. This parameter should be commented out, as it is unnecessary for this configuration since there is a default location of '/' defined.

The second parameter (index) defines the default page that will be served If no page was requested (ie. if www.kennedyfamilyrecipes.com is requested). This parameter should be commented it as we want all dynamic content, including the main page, to be generated by our Flask web application.

The first two parameters (root and index) are included in this configuration file, as they can be useful for some configurations of NGINX.

The third parameter (server_name) and fourth parameter (listen) should be used together. If you have a single web application being served, then you should set these parameters as (note: a port does not need to be specified as it will default to port 80):

```
server {
    …
    Listen 192.241.229.181;
    …
}
```

If you need to want to have requests for blog.kennedyfamilyrecipes.com be served by a different Flask application than the standard www.kennedyfamilyrecipes, then you will need separate 'server' blocks using 'server_name' and 'listen':

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

    . . .

}

server {
    listen 80;
    server_name blog.kennedyfamilyrecipes.com;

    . . .
```

```
}
```

NGINX will always select the 'server_name' that is the best match for the request. This means that a request for 'blog.kennedyfamilyrecipes.com' will be a better match to 'blog.kennedyfamilyrecipes.com' than '*.kennedyfamilyrecipes.com'.

The fifth parameter (charset) defines the specified charset to the "Content-Type" response header field. This value should be set to ''utf-8'.

The first 'location' block defines where NGINX should deliver static content from:

```
location /static {
     alias /usr/src/app/project/static;
 }
```

The location block defines how to process the requested URI (the part of the request that comes after the domain name or IP address/port). In this first location block (/static), we are specifying that NGINX should retrieve files from the '/usr/src/app/project/static' directory on the server when a request comes in for www.kennedyfamilyrecipes.com/static/. For example, a request for www.kennedyfamilyrecipes.com/static/img/img_1203.jpg will come be the picture located at /usr/src/app/project/static/img/img_1203.jpg. If this file does not exist, then the 404 error code (NOT FOUND) will be returned to the user.

The second location block ('/') defines the reverse proxy. This location block defines how NGINX should pass these requests to the WSGI server (Gunicorn) that can interface with our Flask application. Let's look at each parameter in more detail:

```
   location / {
       proxy_pass http://web:8000;
       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
       client_max_body_size 5M;
   }
```

The first parameter (proxy_pass) in this location block defines the location of the proxy server to send the request to. If you just want to pass the request to a local server running on the same machine:

```
proxy_pass http://localhost:8000/;
```

If you want to pass the request to a specific Unix socket, such as when you have NGINX and Gunicorn running on the same server:

```
proxy_pass http://unix:/tmp/backend.socket:/
```

If you are using NGINX as a Docker container that is talking to a Gunicorn container, when you simply need to include the name of the container running Gunicorn:

```
proxy_pass http://web:8000;
```

The second parameter (proxy_pass_header) allows you to redefine the header fields that NGINX sends to the upstream server (ie. Gunciorn). This parameter is used four times to define:

- the name and port of the NGINX server (Host $host)
- the schema of the original client request, as in whether it was an http or an https request (X-Forwarded-Proto $scheme)
- the IP address of the user (X-Real-IP $remote_addr)
- the IP addresses of every server the client has been proxied through up to this point (X-Forwarded-For $proxy_add_x_forwarded_for)

The third parameter (client_max_body_size) defines the maximum size for files being uploaded, which is critical if you web application allows file uploads. Given that image sizes are often 2 MBs in size, a value of 5 MB provides some flexibility to support almost any image.

Conclusion

This blog post described what the NGINX server does and how to configure it for a Flask application. NGINX is a key component to most web applications as it serves static content to the user, reverse proxies requests to an upstream server (WSGI server in our Flask web application), and provides load balancing (not discussed in detail in this blog post). Hopefully, the configuration of NGINX is easier to understand after reading this blog post!

References

How to Configure NGINX (Linode)

NGINX Wiki

NGINX Pitfalls and Common Mistakes

How to Configure the NGINX Web Server on a VPS (DigitalOcean)

Understanding NGINX Server and Location Block Selection Algorithms (DigitalOcean)

NGINX Optimization: Understanding sendfile, tcp_nodelay, and tcp_nopush

Deployment, Flask Tutorial

DEPLOYMENT DOCKER FLASK NGINX

PREVIOUS POST
Why I Switched from a Traditional Deployment to Using Docker

NEXT POST
How to use Docker and Docker Compose to Create a Flask Application

3 Comments

Ryan
FEBRUARY 21, 2017 AT 9:52 PM

/usr/ is the wrong place to put website data.

REPLY

Julian
MAY 8, 2017 AT 6:28 AM

Hi Patrick,

Thanks for the very explanatory post. I'm running a Flask web app on a Nginx server, as well. However, I ran into a problem after securing the website with SSL. Basically, the Flask app is pulling a Bokeh session, that's running on the same machine, and serving it to the web. The problem is that the Bokeh app is served with 'http' and not 'https', which causes the browser to reject it. I assume that the solution lies in the Nginx config file, or the Flask app itself. Here's a more

detailed description of the problem: http://stackoverflow.com/questions/43743029/reverse-proxying-flask-app-with-bokeh-server-on-nginx

Maybe you could provide some suggests that would help me solve this problem.

Thanks!

REPLY

patkennedy79@gmail.com (Post author)
JUNE 23, 2017 AT 5:38 AM

Sorry, but I don't have any experience with Bokeh. Not sure if this recent article from Miguel Grinberg would help, but this is a great article on 'Running your Flask Application over HTTPS'.

REPLY

Leave a Reply

Your email address will not be published.

Name

Email

Website

Post Comment

Search form SEARCH

RECENT POSTS

Structuring a Flask Project

Steps for Starting a New Flask Project using Python3

Using Docker for Flask Application Development (not just Production!)

Software Development Checklist for Python Applications

Python Logging Tutorial

TAGS

administrative API bcrypt Bootstrap Chart.js coverage database Deployment Django docker docker compose documentation email error Family Recipes flask flask-login flask-migrate flask-uploads flask-wtf form git GitHub GitLab Heroku Introduction itsdangerous Javascript logging Moment.js MVC nginx nose2 PostgreSQL python sphinx sql SQLAlchemy template inheritance templates Tutorial Unit Testing Virtual Environment

ARCHIVES

February 2018

June 2017

May 2017

March 2017

January 2017

November 2016

October 2016

September 2016

August 2016

July 2016

June 2016

January 2016

September 2015

August 2015

July 2015

CATEGORIES

Deployment

Family Recipes Website

Flask Tutorial

Introduction

Javascript

Python Guide

Uncategorized

META

Log in

Entries RSS

Comments RSS

WordPress.org