Deploying nginx + django + python 3

Hello,

due to the lacks of informations about deploying latests version of django (1.6+) with latest nginx (1.6+) using gunicorn (18+) inside virtual environment of python 3 (3.4+), it was really hard for a beginner like me to deploy a django project.

I finally made it and now after several months I decided to share my experiences with all the world. So again:

What this guide is about

We'll deploy (that means making website available to the world) `django` project with server engine `nginx` using `gunicorn` for that. We'll also use virtual environment of python and installation will be static - it will not depend on your system-wide python installation.

Prerequisites and informations

I'm using Linux and commands I'm going to introduce are thought to be run in bash shell. Sometimes root privileges might be required and I'll **not** remark that. If you are not familiar with Linux, please read my other guides.

You do not need any special knowledge. But keep in mind that this is not guide how django, nginx or gunicorn work! This is about how it should be brought together to work.

My choice - why nginx, python 3 etc.

I'm not the one who tried all of possible choices. I've just tried a lot of them and this one was first of them which worked. So I stuck to it.

I've chosen `nginx` over `apache` because this seems to be *trend* today thanks to Apache's age. It's also seems to me easier.

I've chosen `django` because I love python. And I'm quite new to it so when I decided to learn this great language, I started with `python 3`. It was somehow logical because I could choose what I want and the newer one is of course better investment to the future.

Gunicorn is just so easy to use and I've found great documentations and guides for it.

Virtual environment is necessity. You don't want all python projects (not just django websites) depend on one specific configuration. It's not stable (one package can hurt other) nor secure. We'll use **hard** *virtualenv* because it's also safer - you can update this version when you want and not with every time your distribution says to you.

How the hell all that works

Here is a little model I've made for myself and I think it's not too bad to be a starting point for you ;) . We

have five terms: `nginx`, `python`, `virtualenv`, `gunicorn` and `django`.

First layer (nginx)

`nginx` is what cares about requests from the world. It's what catches your request (e.g. Google) and redirects it to according folder (in case of static HTML page with index.html, not our case), or to some application.

Second layer (gunicorn)

This application in our case is `gunicorn`. It's powered by `python` and it basically makes a magical communication channel `nginx``~``django app`. This tunnel is represented by **socket** (we'll get to it). Why can't do this `nginx`? It's just not clever enough (better - it just hasn't do that and that's absolutely OK in Unix philosophy). Gunicorn can make *server* similar to django test server. But it can also **serve** django app content to `nginx` and hence solving nginx's limitation.

Third layer (django)

Then there is just `django` - your project with you pages - this is what your website is about. All previous (and next) is just *working* layer.

Wrapper for second and third layer

`python` is *engine* of `gunicorn` even for `django`. This `python` is running inside *sandbox* called virtualenv. Gunicorn will **activate** this virtualenv for us and *run* django app.

That's all. Not that hard, huh? Basically **nginx~gunicorn~django**. `python` is powering it and `virtualenv` is just wraps python version (it is not necessity to have virtualenv, if it's easier to understand it).

Let's do it

nginx configuration

Covered in other tutorial, you must be able to make it to the point that you are able to see *nginx welcome page*.

Installing python and virtualenv

About virtual environments there is tons of guides on the internet - feel free to educate yourself :D . Here is my brief guide.

To do that we'll need to install `python-virtualenv`. Install it and then create some folder for our test case. Let it be `/var/www/test`. Also install `python` of version 3, if you haven't done that before.

We'll now create *sandbox* of python for our test case inside this folder (`/var/www/test`). Why `/var/www`? It's just good place to put websites on Unix (and hence - Linux). But it can be anywhere of course. To create a **REAL** copy (and not just a linked variant) of python version 3 we need to use this syntax:

```
virtualenv --python=python3 --always-copy venv
```

What happened here? We created python *sandbox* called **venv** in current directory. It use **python3** as default choice and we copied all necessary files for life of this installation (by default there are only symlinked to the system one).

What now? We need to switch from *system* python installation to *venv* python installation. First try to type

`python -c "import sys; print(sys.path)"`. The output is similar to this:

```
['', '/usr/lib/python34.zip', '/usr/lib/python3.4', '/usr/lib/python3.4/plat-linux', '/usr/lib/python3
```

where you can notice that current default `python` interpreter gets it's config from somewhere in `/usr/lib/...`.

We will now activate our virtualenv by this command: `source /var/www/test/venv/bin/activate`. Now try same command as above (`python -c ...`) and it should print instead of `/usr/lib/...` something starting with `/var/www/test/venv/...`. If yes, it's working :) .

To quit from this environment and get to your system-wide, type `deactivate`.

pip installing django and gunicorn

One of the best advantages of `python 3.4` is a fact that `pip` is installed by default. What is `pip`? `pip` is installer for python packages.

All python packages can be found here. Of course you can find you package there (try for example with `django`), download it and build it with python on your own. But that sound like a lot of work. Let's `pip` do it for us.

Assure yourself you are working in our virtualenv (you can again **activate** it) and type this:

```
pip install django
pip install gunicorn
```

you can specify version just by typing `=` behind name package:

```
pip install django=1.6.0
```

but of course this version must exists on `pypi.python.org`. If there are errors, try adding `-v` switch for verbose.

To list installed packages type:

```
pip list
```

try if you see `django` and `gunicorn` there :) .

and that's all you need for now with pip (although there isn't much more about pip).

Sample django project

Now we'll need to create django project for our test case. Go inside `/var/www/test` and activate our virtualenv where is `django` and `gunicorn` (you can do that again by `source /var/www/test/venv/bin/activate`).

Create django project by:

```
django-admin3.py startproject ourcase
```

it should create this structure inside `/var/www/test`:

```
ourcase
|-- manage.py
`-- ourcase
    |-- __init__.py
    |-- settings.py
    |-- urls.py
    `-- wsgi.py

1 directory, 5 files
```

check if it's working with local django testing server by `python manage.py runserver`. Check in browser `127.0.0.1:8000` - if there is django welcome page, it's good.

Just for comfort make `manage.py` executable by `chmod +x ourcase/manage.py`.

gunicorn and daemonizing it

Now we'll replace django testing server, which is just for kids (it's just great future :)), with fully mature nginx for adults.

As was previously stated, for that we'll need gunicorn. Gunicorn will have to be running to enable communication between nginx and django project.

First, we'll use just `gunicorn` to display our django test project on `127.0.0.1:8000`. It's incredibly easy. Again - assure yourself you are working in current virtualenv.

Now navigate yourself inside `/var/www/test/ourcase/` and run this magical command:

```
gunicorn ourcase.wsgi:application
```

it will start something like *gunicorn server* - you should be able to see your django welcome page on `127.0.0.1:8000`.

This is just the most stupid configuration, which is enough for this test, but not for deploying on server. For that we'll want to add much more. Create starting script `/var/www/test/gunicorn_start.sh`:

```
#!/bin/bash

NAME="ourcase"                                  #Name of the application (*)
DJANGODIR=/var/www/test/ourcase                 # Django project directory (*)
SOCKFILE=/var/www/test/run/gunicorn.sock        # we will communicate using this unix socket (*)
USER=nginx                                      # the user to run as (*)
GROUP=webdata                                   # the group to run as (*)
NUM_WORKERS=1                                   # how many worker processes should Gunicorn spawn (*
DJANGO_SETTINGS_MODULE=ourcase.settings         # which settings file should Django use (*)
DJANGO_WSGI_MODULE=ourcase.wsgi                 # WSGI module name (*)

echo "Starting $NAME as `whoami`"

# Activate the virtual environment
cd $DJANGODIR
source /var/www/test/venv/bin/activate
export DJANGO_SETTINGS_MODULE=$DJANGO_SETTINGS_MODULE
export PYTHONPATH=$DJANGODIR:$PYTHONPATH

# Create the run directory if it doesn't exist
RUNDIR=$(dirname $SOCKFILE)
test -d $RUNDIR || mkdir -p $RUNDIR

# Start your Django Unicorn
# Programs meant to be run under supervisor should not daemonize themselves (do not use --daemon)
exec /var/www/test/venv/bin/gunicorn ${DJANGO_WSGI_MODULE}:application \
  --name $NAME \
  --workers $NUM_WORKERS \
  --user $USER \
  --bind=unix:$SOCKFILE
```

Wow! A lot happened here compared to our *stupid* variant. Everything marked with `(*)` in comments can be changed (or must be changed if your paths differs).

The most important change here is that we added `SOCKFILE` - socket. This is the magic thingie which will enable `nginx` to server django project (app). Gunicorn will somehow run server as in previous stupid variant and *transfer* this into socket file in language which `nginx` understands. `nginx` is looking to this socket file and is happy to serve everything there is.

It's common practice (and I strongly encouraged it) to run server as some specific user. It's for security reasons. So if you haven't done it before, create some user and group for these purposes (ALSO IN OTHER MY TUTORIAL).

Workers are just how much computing power you enable for this website.

If you are not working as a user which is in script set to `USER` variable, you **won't** be able to run this script (you'll get some errors). That's because of permissions reasons. If you'd like to check or debug this script (and it's recommended), uncomment `--user $USER` line - it should work then even if you run it as another user. Of course you need to make script executable.

See gunicorn documentation for more informations.

This script is laying all over the internet in multiple variants. If you have problems to run it, try to uncomment some other lines in last part of script. For example I wasn't able to run this script with directive `--log-level=warning`.

If it is working, it's great! Now we'll daemonize it by using `systemd`. Of course you can use another init system (like Ubuntu `upstart`. Just search for "how to run script after boot".

Create new service file `/usr/lib/systemd/system/gunicorn_ourcase.service` and insert this:

```
[Unit]
Description=Ourcase gunicorn daemon

[Service]
Type=simple
User=nginx
ExecStart=/var/www/test/gunicorn_start.sh

[Install]
WantedBy=multi-user.target
```

now enable it as with other units:

```
systemctl enable gunicorn_ourcase
```

now this script should be run after boot. Try if it's working (reboot and use `systemctl status gunicorn_ourcase`).

That's all for gunicorn.

django project deployment

Deploying django project is topic for longer tutorial then is this. So I'll make it as small as possible.

If you've just developed django project with test server, it makes a tons of things for you without any notices. In reality it's not as easy - everything isn't done automatically and django is prepared for that - but you need to *activate* this futures, since it's not by default.

Directories

Nice example is with **static** files. There are e.g. some CSS styles for django administration page. These needs to be in special folder and we'll tell nginx that when website asks for file `style.css`, it should looks into `~/var/www/test/ourcase/static/style.css`.

But how to find all this static files? Right now they are sourced from django installation directory (probably something like `/var/www/test/venv/lib/python3.4/django/...`. `manage.py` has a special command for this, but first we need to tell him few details in `settings.py`.

The most common configuration is to has a special directory for static files where you can edit them, past them etc. Then there will be static directory, where you won't do any changes - this will be for `manage.py` command - it will collects them from your special directory, from django installation directory etc. In templates, when you want to use e.g. some static image on background, you use `{ STATIC_URL}/static_images/mybgrnd.png`.

To do this we'll add this to `settings.py`:

```
STATIC_URL = '/static/'
STATIC_ROOT = os.path.join(BASE_DIR, "static")
STATICFILES_DIRS = (os.path.join(BASE_DIR, "sfiles"), )
```

all your static files used should now be placed inside `/var/www/test/ourcase/sfiles`. If you just want to try it, create this directory and `touch sfiles/example.png` inside it.

Now run `./manage.py collectstatic`. It should ask you if you really want to do that (and you want). Process will start and after it's finish you'll have collected all static files inside `static` folder. This you need to do every time you change something inside `sfiles` folder.

Websites also usually has `media` folder, which is used for user files - for example images to blog posts. Usually we use `MEDIA_URL` for calling things from media dir in templates.

Configuration should be same as with django testing server and you don't need to do any special changes here. My looks like this:

```
MEDIA_ROOT = os.path.join(BASE_DIR, "media")
MEDIA_URL = '/media/'
ADMIN_MEDIA_PREFIX = '/media/admin/'
```

and all user files (uploaded images, sounds...) are inside */var/www/test/ourcase/media`* directory. You don't need to do something like `collectstatics` here.

Steps for other directories should be same.

Enough for directories. But some other changes are needed to deploy django project. In some cases I don't really know why I need to add this to `settings.py`, but I know what that does and it's just working.

Templates

I had to add this for templates:

```
TEMPLATE_DIRS = (os.path.join(BASE_DIR, 'templates'),)
TEMPLATE_LOADERS = (
'django.template.loaders.filesystem.Loader',
'django.template.loaders.app_directories.Loader',)
```

where I've put my `base.html` which is used in all other templates in whole website (in every app). If you use `flatpages`, you can also make a directory inside `templates` called `flatpages`, where you can copy `base.html` as `default.html` and use this template as base for flatpages.

SITE_ID

For some purposes is needed to set SITE_ID. In my case it was because of `flatpages`. It's easy:

```
SITE_ID = 1
```

ALLOWED_HOSTS

You need to past all your domains here. If your domain is `www.example.com` and I guess `example.com` also, it should looks like this:

```
ALLOWED_HOSTS = ['example.com', 'www.example.com']
```

DEBUG

This directive should be set to **False**. But when you are configuring your server for first time, let **True** there. It helps you find out bugs on your site.

That's it!

nginx server configuration

Last part is configuring `nginx` to make him listen on socket created by gunicorn. It's not hard.

Edit `/etc/nginx/nginx.conf` and paste this into `http` block:

```
upstream test_server {
  server unix:/var/www/test/run/gunicorn.sock fail_timeout=10s;
}

# This is not neccessary - it's just commonly used
# it just redirects example.com -> www.example.com
# so it isn't treated as two separate websites
server {
        listen 80;
        server_name example.com;
        return 301 $scheme://www.example.com$request_uri;
}

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

    client_max_body_size 4G;

    access_log /var/www/test/logs/nginx-access.log;
    error_log /var/www/test/logs/nginx-error.log warn;

    location /static/ {
        autoindex on;
        alias   /var/www/test/ourcase/static/;
    }

    location /media/ {
        autoindex on;
        alias   /var/www/test/ourcase/media/;
    }

    location / {
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $http_host;
        proxy_redirect off;

        if (!-f $request_filename) {
            proxy_pass http://test_server;
            break;
        }
    }

    #For favicon
    location  /favicon.ico {
        alias /var/www/test/test/static/img/favicon.ico;
    }
    #For robots.txt
    location  /robots.txt {
        alias /var/www/test/test/static/robots.txt ;
    }
    # Error pages
    error_page 500 502 503 504 /500.html;
    location = /500.html {
        root /var/www/test/ourcase/static/;
    }
}
```

OK, that's whipping. I'll be fast.

First, we tell *nginx* where is socket file (`gunicorn.sock`) from gunicorn.

Then there is redirect from non-www domain to www domain. This can be omitted or solved in other way (CNAME).

Then there is main body of server configuration: * logs are useful for catching bugs and errors - has multiple parameters, like how much should they bother you. Don't forget to create log directory. * static and media block - these are extremely important - this is why we played all that games with collectstatics etc. It just tells nginx where it should looks when website asks for e.g. `/static/style.css/` or `/media/img/picture_of_my_cat.png`. * Block with all that proxy things is also important and is used for technical background around socket communication and redirecting. Don't care about that. * Favicon and robots.txt is not necessary, but all browsers and web crawlers are still searching for them. So if you don't like errors in your logs, add create these two things. * Last block is telling nginx where it should looks for error pages when something doesn't exists.

Save and exit. Next great future of `nginx` is it's ability of checking configuration. Type `nginx -t` (don't forget root permissions) and you'll see if configuration is syntactically correct. Don't forget about that stupid `;`.

Finally enable `nginx` to be ran after reboot:

```
systemctl enable nginx
```

Some sugar candy

Install with `pip` package called `setproctitle`. It's useful for displaying more info about ran processes like gunicorn in system process managers (`htop`, `ps`, ...).

Debugging

That's it. Now restart computer and see if it doesn't explode. You can analyse `nginx` or `gunicorn` with `systemctl`, e.g.:

```
systemctl status gunicorn_ourcase
```

and some informations should be also in log files. Try to get to your website from browser and see what happens. Don't forget that browser likes caching and press **CTRL+r** for reload to see changes you've made.

After every change in configuration of nginx you need to restart it by running `nginx -s reload`.

To see what processes are spawned you can use your task manager like `htop` or `ps`.

Integration with GitHub

For starter it's necessary to say, that GitHub...

...is awesome! If you don't needed and you went through whole process above, you can probably save a lot of headaches just by using GitHub. You don't need any special knowledge for start, but you will need to learn them on the fly while reading this tutorial (there is really a lot about git out there, google is your friend).

The variant I propose here is very easy, scalable and fast. Probably the most easy and effective I've found.

Why to use it

I was so happy when I deployd my first django project. But few weeks later I've found that it's just not feeling right to make changes on live version of the website (sometimes refered as *production*). So I started to use GitHub and found a solution.

Here I will cover this: For every your website you end up with one directory including three subdirectories.

1. First called **production** - it's the one which is live on the internet - the one what `nginx` refers.
2. Second called **mydomain.git** - this one is necessary for our github configuration. You will barely change there anything
3. Last one - **work_dir** - the one where all changes are being made and is connected to GitHub

Workflow

Your work will look like this:

1. Your work_dir contains master branch. This branch can be pushed to production (to go live) anywhen! So when you want to make change to your website, you need create new branch (correctly named based on the change you are doing - e.g. *hotfix_plugin*, *typo_css*...) and when you finish and test this

branch, you merge it to master.
2. You push master to your GitHub repository
3. You push master to your production folder on your computer

Set it all up

So how to do it? I suppose you have one working directory as we created in previous chapters.

Now go to the place where you websites are stored. Mine is in `/var/www` and create this structure:

```
mydomain
├── mydomain.git
├── production
└── work_dir
```

Go to `/var/www/mydomain.git` and type this:

```
git init --bare
```

this will create just git repository with some special folders. You don't need to know anything about it. All you need to do is to create this file `/var/www/mydomain/mydomain.git/hooks/post-receive` and add this:

```
#!/bin/sh
git --work-tree=/var/www/mydomain/production --git-dir=/var/www/mydomain/mydomain.git checkout -f
```

and make the script runable `chmod +x /var/www/mydomain/mydomain.git/hooks/post-receive`

Go to work_dir and paste there you current *production* code (the one from previous chapters). Now you need to make a GitHub repository from that. The best guide is this one: How to add existing folder to GitHub. (Maybe you'll need to generate SSH key). Is it working? Great.

> Note: It's very good to make git repositories as small as possible, so don't add to repository files which are not necessary or you backup them somewhere else. But virtualenv is a good thing to add there to IMHO.

Now just add another remote, which will point to our created git repository. Every time we'll want to go live with master, we'll push changes to this git repository and it will take care to transfer our files to production. So in work_dir type:

```
git remote add production file:///var/www/mydomain/mydomain.git``
```

and that's all. When you now want to push changes to production, type `git push production master`. Congratulations!

Finalization

That's all! I hope this guide helped you and you has successfully start up your websites! :)

For further reading, I'd recommend you to look at the part how to set up git for easy deployment.

Previous Next

Built with Sphinx using a theme provided by Read the Docs.