Data hacker | Entrepreneur | Engineer | Kitesurfer | https://codingstartup.com/

Aug 28, 2017 · 13 min read

Deploy Django app with Nginx, Gunicorn, PostgreSQL & Supervisor

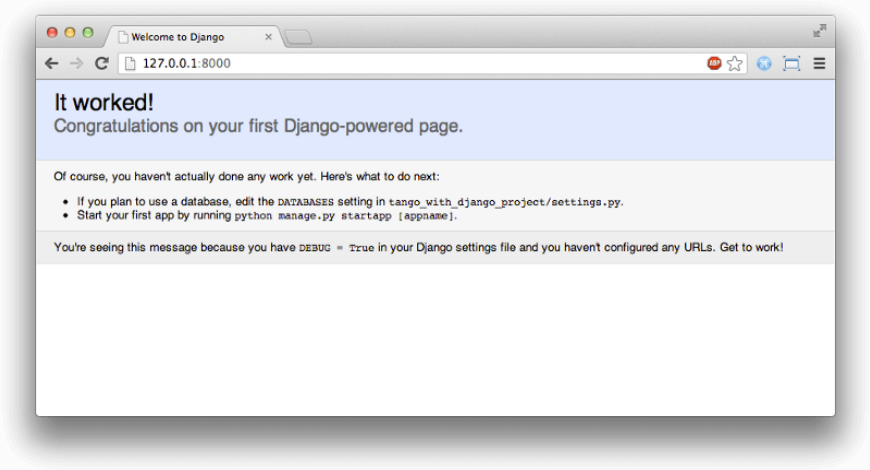
Django is the most popular Python-based web framework for a while now. Django is powerful, robust, full of capabilities and surrounded by a supportive community. Django is based on models, views and templates, similarly to other MVC frameworks out there.

Django provides you with a development server out of the box once you start a new project using the commands:

```
$ django-admin startproject my_project
$ python ./manage.py runserver 8000
```

With two lines in the terminal, you can have a working development server on your local machine so you can start coding. One of the tricky parts when it comes to Django is deploying the project so it will be available from different devices around the globe. As technological entrepreneurs, we need to not only develop apps with backend and frontend but also deploy them to a production environment which has

to be modular, maintainable and of course secure.



Deployment of a Django app requires different mechanisms which will be listed. Before we begin, we need to perform an alignment in terms of the tools we are going to use throughout this post:

Python version 2.7.6

Django version 1.11

Linux Ubuntu server hosted on DigitalOcean cloud provider

Linux Ubuntu local machine

Git repository containing your codebase

I assume you are already using 1, 2, 4 and 5. About the Linux server, we are about to create it together during the first step of the deployment tutorial. Please note that this post discusses deployment on a single Ubuntu server. This configuration is great for small projects, but in order to scale your resources up to support larger amounts of traffic, you should consider a high-availability server infrastructure, using load balancers, floating IP addresses, redundancy and more.

Linux is much more popular for serving web apps than Windows. Additionally, Python and Django work together very well with Linux, and not so well with Windows.

There are many reasons for choosing DigitalOcean as a cloud provider, especially for small projects that will be deployed on a single droplet (a virtual server in DigitalOcean terminology). DigitalOcean is a great solution for software projects and startups which start small and scale

up step by step. Read more about my comparison between DigitalOcean and Amazon Web Services in terms of an early-stage startup software project.

There are some best practices for setting up your Django project I highly recommend you to follow before starting the deployment process. The best practices include working with a virtual environment, exporting requirements.txt file and configuring the settings.py file for working with multiple environments.

This post will cover the deployment process of a Django project from A to Z on a brand-new Linux Ubuntu server. Feel free to choose your favorite cloud provider other than DigitalOcean for deployment.

As aforesaid, the built-in development server of Django is weak and is not built for scale. You can use it for developing your Django project yourself or share it with your co-workers, but not more than that. In order to serve your app in a production environment, we need to use several components that will talk to each other and make the magic happen. Hosting a web application usually requires the orchestration of three actors:

- Web server
- Gateway
- Application

The web server

The web server receives an HTTP request from the client (the browser) and is usually responsible for load balancing, proxying requests to other processes, serving static files, caching and more. The web server usually interprets the request and sends it to the gateway. Common web server and Apache and Nginx. In this tutorial, we will use Nginx (which is also my favorite).

The Gateway

The gateway translates the request received from the web server so the application can handle it. The gateway is often responsible for logging and reporting as well. We will use Gunicorn as our Gateway for this tutorial.

The Application

As you may already guess, the application refers to your Django app. The app takes the interpreted request, process it using the logic you implemented as a developer, and returns a response.

. . .

2. Assuming you have an existing ready-for-deployment Django project, we are going to deploy your project by following these steps:

    1. Creating a new DigitalOcean droplet
    2. Installing pre requisites: pip, virtual environment, and git
    3. Pulling the Django app from Git
    4. Setting up PostgreSQL
    5. Configuring Gunicorn with Supervisor
    6. Configuring Nginx for listening to requests
    7. Securing your deployed app: setting up firewall

Creating a droplet

A droplet in DigitalOcean refers to a virtual Linux server with CPU, RAM and disk space. The first step in this tutorial is about creating a new droplet and connect to it via SSH. Assuming your local machine is running Ubuntu, we are going to create a new SSH key pair in order to easily and securely connect to our droplet once it is created. Connection using SSH keys (rather than a password) is both more simple and secure. If you already have an SSH key pair, you can skip the creation process. On your local machine, enter in the terminal:

```
$ ssh-keygen -t rsa
```

You should get two more questions, where to locate the keys (the default is fine) and whether you want to set up a password (not essential).

Now the key pair is located in:

```
/home/user/.ssh/
```

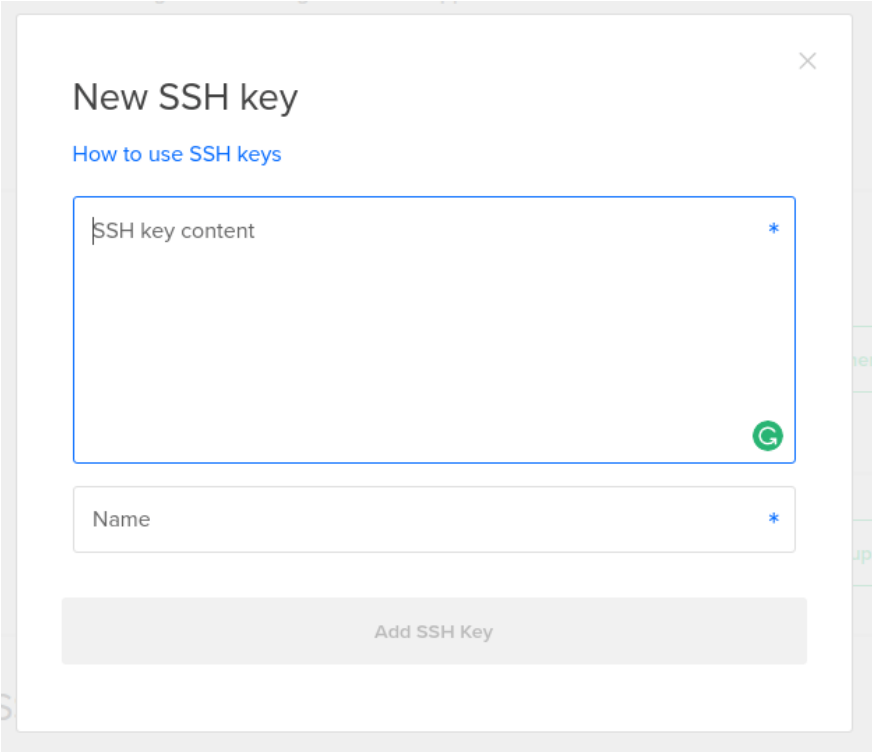
where `id_rsa.pub` is your public key and `id_rsa` is your private key. In order to use the key pair to connect to a remote server, the public key should be located on the remote server and the private key should be located on your local machine.

Notice that the public key can be located on every remote server you wish to connect to. But, the private key must be kept only on your local machine! Sharing the private key will enable other users to connect to your server.

After signing up with DigitalOcean, open the SSH page and click on the Add SSH Key button. In your terminal copy the newly-created public key:

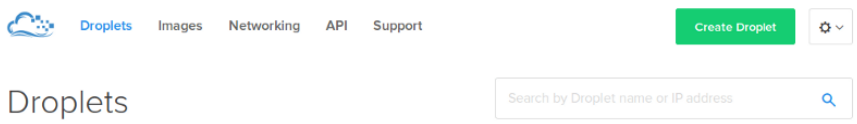
```
$ cat /home/user/.ssh/id_rsa.pub
```

Enter the new public key you generated and name it as you wish.

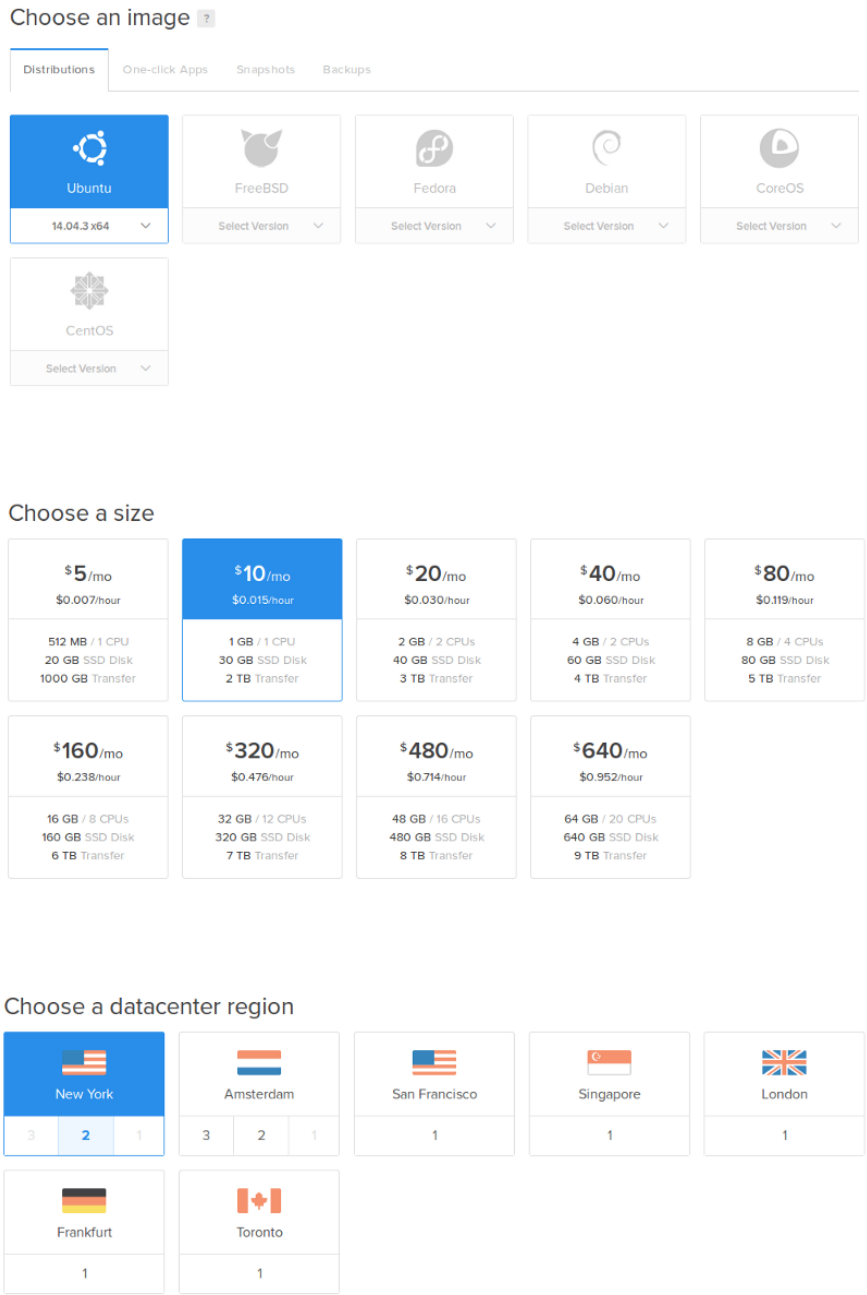


Now once the key is stored in your account, you can assign it with every droplet you create. The droplet will contain the key so you connect to it from your local machine, while password authentication will be disabled by default, which is highly recommended.

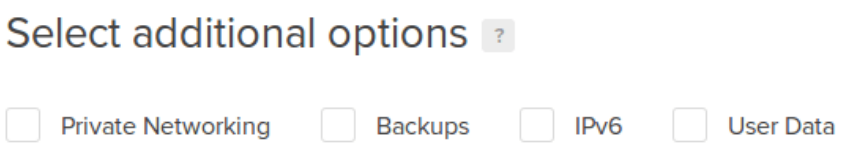
Now we are ready to create our droplet. Click on “Create Droplet” at the top bar of your DigitalOcean dashboard.

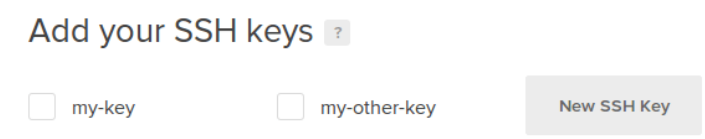


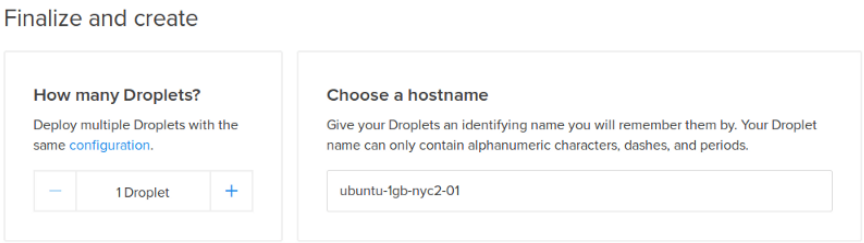
Choose Ubuntu 16.04 64bit as your image, droplet size which is either 512MB RAM or 1GB, whatever region that makes sense to you.



You can select the private networking feature (which is not essential for this tutorial). Make sure to select the SSH key you've just added to your account. Name your new droplet and click "Create".







Once your new droplet has been created, you should be able to connect to it easily using the SSH key you created. In order to do that, copy the IP address of your droplet from your droplets page inside your dashboard, go to your local terminal and type:

```
$ ssh root@IP_ADDRESS_COPIED
```

Make sure to replace with `IP_ADDRESS_COPIED` your droplet's IP address. You should be already connected by now.

Tip for advanced users: in case you want to configure an even simpler way to connect, add an alias to your droplet by editing the file:

```
$ nano /home/user/.ssh/config
```

and adding:

```
Host remote-server-name
    Hostname DROPLET_IP_ADDRESS
    User root
```

Make sure to replace `remote-server-name` with a name of your choice, and `DROPLET_IP_ADDRESS` with the IP address of the server.

Save the file by hitting `Ctrl+O` and then close it with `Ctrl+X` . Now all you need to do in order to connect to your droplet is typing:

```
$ ssh remote-server-name
```

That simple.

Installing prerequisites

Once connected to your droplet, we are going to install some software in order to start our deployment process. Start by updating your repositories and installing pip and virtualenv.

```
$ sudo apt-get update $ sudo apt-get install python-pip
python-dev build-essential libpq-dev postgresql postgresql-
contrib nginx git virtualenv virtualenvwrapper
$ export LC_ALL="en_US.UTF-8"
$ pip install --upgrade pip
$ pip install --upgrade virtualenv
```

Hopefully, you work with a virtual environment on your local machine. In case you don't, I highly recommend you reading my best practices post for setting up a Django project in order to realize why working with virtual environments is an essential part of your Django development process.

Let's get to configuring the virtual environment. Create a new folder with:

```
$ mkdir ~/.virtualenvs
$ export WORKON_HOME=~/.virtualenvs
```

Configure the virtual environment wrapper for easier access by running:

```
$ nano ~/.bashrc
```

and adding this line to the end of the file:

```
. /usr/local/bin/virtualenvwrapper.sh
```

Tip: use `Ctrl+V` to scroll down faster, and `Ctrl+Y` to scroll up faster inside the nano editor.

Hit `Ctrl+O` to save the file and `Ctrl+X` to close it. In your terminal type:

```
$ . .bashrc
```

Now you should be able to create your new virtual environment for your Django project:

```
$ mkvirtualenv virtual-env-name
```

From within your virtual environment install:

```
(virtual-env-name) $ pip install django gunicorn psycopg2
```

Tip: Useful command for working with your virtual environment:

```
$ workon virtual-env-name # activate the virtual environment
$ deactivate              # deactivate the virtual
environment
```

Pulling application from Git

Start by creating a new user that will hold your Django application:

```
$ adduser django
$ cd /home/django
$ git clone REPOSITORY_URL
```

Assuming your code base is already located in a Git repository, just type your password and your repository will be cloned into your remote server. You might need to add permissions to execute `manage.py` by navigating into your project folder (the one you've just cloned) and type:

```
$ chmod 755 ./manage.py
```

In order to take the virtual environment one step further in terms of simplicity, copy the path of your project's main folder to the virtual environment settings by typing:

```
$ pwd > /root/.virtualenvs/virtual-env-name/.project
```

Make sure to replace `virtual-env-name` with the real name of your virtual environment. Now, once you use the `workon` command to activate your virtual environment, you'll be navigated automatically to your project's main path.

In order to setup the the environment variable properly, type:

```
$ nano /root/.virtualenvs/virtual-env-name/bin/postactivate #
replace virtual-env-name with the real name
```

and add this line to the file:

```
export DJANGO_SETTINGS_MODULE=app.settings
```

Make sure to replace `app.settings` with the location of your settings module inside your Django app. Save and close the file.

Assuming you've set up your `requirements.txt file` as described in the Django best practices post, you're now able to install all your requirements at once by navigating to the path of the `requirements.txt` file and run from within your virtual environment:

```
(virtual-env-name) $ pip install -r requirements.txt
```

Setting up PostgreSQL

Assuming you've set up your settings module as described in the Django best practices post, you should have by now a separation between the development and production settings files. Your `production.py` settings file should contain PostgreSQL connection settings as well. If it doesn't, add to the file:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'app_db',
        'USER': 'app_user',
        'PASSWORD': 'password',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

I highly recommend updating and pushing the file on your local machine and pulling it from the remote server using the repository we cloned.

Let's get to creating the production database. Inside the terminal, type:

```
$ sudo -u postgres psql
```

Now you should be inside PostgreSQL terminal. Create your DB and user with:

```
> CREATE DATABASE app_db;
> CREATE USER app_user WITH PASSWORD 'password';
> ALTER ROLE app_user SET client_encoding TO 'utf8';
> ALTER ROLE app_user SET default_transaction_isolation TO
'read committed';
> ALTER ROLE app_user SET timezone TO 'UTC';
> ALTER USER app_user CREATEDB;
> GRANT ALL PRIVILEGES ON DATABASE app_db TO app_user;
```

Make sure your details here match the `production.py` settings file DB configuration as described above. Exit the PostgreSQL shell by typing `\q` .

Now you should be ready to run migrations command on the new DB. Assuming all of your migrations folders are in the `.gitignore` file, meaning they are not pushed into the repository, your migrations folders should be empty. Therefore, you can set up the DB by navigating to your main project path with:

```
(virtual-env-name) $ cdproject
```

and then run:

```
(virtual-env-name) $ python ./manage.py migrate
(virtual-env-name) $ python ./manage.py makemigrations
(virtual-env-name) $ python ./manage.py migrate
```

Don't forget to create yourself a superuser by typing:

```
(virtual-env-name) $ python ./manage.py createsuperuser
```

Configuring Gunicorn with Supervisor

Now once the application is set up properly, it's time to configure our gateway for sending requests to our Django application. We will use Gunicorn as our gateway, which is commonly used.

Start by navigating to your project's main path by typing:

```
(virtual-env-name) $ cdproject
```

First, we will test gunicorn by typing:

```
(virtual-env-name) $ gunicorn --bind 0.0.0.0:8000
app.wsgi:application
```

Make sure to replace `app` with your app's name. Once gunicorn is running your application, you should be able to access `http://IP_ADDRESS:8000` and see your application in action.

When you're finished testing, hit `Ctrl+C` to stop gunicorn from running.

Now it's time to operate gunicorn from a service to make sure it's running continuously. Rather than setting up a `systemd` service, we will use a more robust way with Supervisor. Supervisor, as the name suggests, is a great tool for monitoring and controlling processes. It helps you understand better how your processes operate.

To install supervisor, type outside of your virtual environment:

```
$ sudo apt-get install supervisor
```

Once supervisor is running, every `.conf` file that is included in the path:

```
/etc/supervisor/conf.d
```

represents a monitored process. Let's add a new `.conf` file to monitor gunicorn:

```
$ nano /etc/supervisor/conf.d/gunicorn.conf
```

and add into the file:

```
[program:gunicorn]
directory=/home/django/app-django/app command=/root
/.virtualenvs/virtual-env-name/bin/gunicorn --workers 3
--bind unix:/home/django/app-django/app/app.sock
app.wsgi:application
autostart=true
autorestart=true
stderr_logfile=/var/log/gunicorn/gunicorn.out.log
stdout_logfile=/var/log/gunicorn/gunicorn.err.log
user=root
group=www-data
environment=LANG=en_US.UTF-8,LC_ALL=en_US.UTF-8

[group:guni]
programs:gunicorn
```

Make sure that all the references are properly configured. Save and close the file.

Now let's update supervisor to monitor the gunicorn process we've just created by running:

```
$ supervisorctl reread $ supervisorctl update
```

In order to validate the process integrity, use this command:

```
$ supervisorctl status
```

By now, gunicorn operates as an internal process rather than a process that can be accessed by users outside the machine. In order to start sending traffic to gunicorn and then to your Django application, we will set up Nginx the serve as a web server.

Configuring Nginx

Nginx is one of the most popular web servers out there. The integration between Nginx and Gunicorn is seamless. In this section, we're going to set up Nginx to send traffic to Gunicorn. In order to do that, we will create a new configuration file (make sure to replace `app` with your own app name):

```
$ nano /etc/nginx/site-available/app
```

then edit the file by adding:

```
server {
   listen 80;
    server_name SERVER_DOMAIN_OR_IP;
    location = /favicon.ico { access_log off; log_not_found
off; }
    location /static/ {
        root /home/django/app-django/app;
    }
    location / {
        include proxy_params;
        proxy_pass http://unix:/home/django/app-django
/app/app.sock;
    }
}
```

This configuration will proxy requests to the appropriate route in your server. Make sure to set all the references properly according to Gunicorn and to your app configurations.

Initiate a link with:

```
$ ln -s /etc/nginx/sites-available/app /etc/nginx/sites-enabled
```

Check Nginx configuration by running:

```
$ nginx -t
```

Assuming all good, restart Nginx by running:

```
$ systemctl restart nginx
```

By now you should be able to access your server only by typing your IP in the browser because Nginx listens on port 80 which is the default port browsers use.

Security

Well done! You should have a deployed Django app by now! Now it's time to secure the app to make sure it's much difficult to hack it. In order to do that, we will use `ufw` built-in Linux firewall.

ufw works by configuring rules. Rules tell the firewall which kind of traffic it should accept or decline. At this point, there are two kinds of traffic we want to accept, or in other words, two ports we want to open:

port 80 for listening to incoming traffic via browsers

port 22 to be able to connect to the server via SSH.

Open the port by typing:

```
$ ufw allow 80
$ ufw allow 22
```

then enable `ufw` by typing:

```
$ ufw enable
```

Tip: before closing the terminal, make sure you are able to connect via SSH from another terminal to so you're not locked outside your droplet due to bad configurations of the firewall.

What to do next?

This post is the ultimate guide to deploy a Django app on a single server. In case you're developing an app that should serve larger amounts of traffic, I suggest you look into highly scalable server architecture. You can start with my post about how to design a high-availability server architecture.

. . .

Any comments with additional tips or new points of view would be highly appreciated!

For more insightful posts for coders with entrepreneurial mindset, visit https://codingstartups.com/

Published originally here.