How To Install and Configure Django with Postgres, Nginx, and Gunicorn

Authored by: ***ASPHostServer Administrator*** *[asphostserver@gmail.com]*
Saved From: http://faq.asphosthelpdesk.com/article.php?id=251

For convenience, I've broken this tutorial into two parts. The first part (steps 1 - 6) covers installation only. If you are a more advanced Django user who just needs help getting things installed you can stop at step 6. If you already have everything installed and just want to know how to configure everything, then skip ahead to step 7. If you feel like you need help from start to finish, then just go through the steps in order and you should have no problems. Let's get started!

Step One: Update Packages

Before we do anything, we need to make sure that all the packages installed on our server are up to date. In order to do this, connect to your desired server via SSH and run the following commands:

```
sudo apt-get update
```

```
sudo apt-get upgrade
```

The first command downloads any updates for packages managed through apt-get. The second command installs the updates that were downloaded. After running the above commands if there are updates to install you will likely be prompted to indicate whether or not you want to install the updates. If this happens, just type "y" and then hit "enter" when prompted.

Step Two: Install and Create Virtualenv

Installing virtualenv is very simple. Just run the command below:

```
sudo apt-get install python-virtualenv
```

That's all there is to it! Now let's create our virtualenv so we can install Django and other Python packages within it:

```
sudo virtualenv /opt/myenv
```

Notice that a new directory "myenv" was created in the "/opt" directory. This is where our virtualenv will live. Make sure to replace "/opt/myenv" with the path to where you want your virtualenv installed. I typically put my env's in /opt, but this is strictly preference. Some people create a directory named "webapps" at the root of the server. Choose whatever method makes the most sense to you.

Step Three: Install Django

Now we need to activate our virtualenv so that when we install Python packages they install to our virtualenv.

This is how you activate your virtualenv:

```
source /opt/myenv/bin/activate
```

You should now see that "(myenv)" has been appended to the beginning of your terminal prompt. This will help you to know when your virtualenv is active and which virtualenv is active should you have multiple virtualenv's on the server.

With your virtualenv active, we can now install Django. To do this, we will use pip, a Python package manager much like easy_install. Here is the command you will run:

```
pip install django
```

You now have Django installed in your virtualenv! Now let's get our database server going.

Step Four: Install PostgreSQL

Most Django users prefer to use PostgreSQL as their database server. It is much more robust than MySQL and the Django ORM works much better with PostgreSQL than MySQL, MSSQL, or others.

Since we don't need our virtualenv active for this part, run the following command to deactivate:

```
deactivate
```

This will always deactivate whatever virtualenv is active currently. Now we need to install dependencies for PostgreSQL to work with Django with this command:

```
sudo apt-get install libpq-dev python-dev
```

Now that you have done this, install PostgreSQL like so:

```
sudo apt-get install postgresql postgresql-contrib
```

PostgreSQL is now installed on your machine and ready to roll.

Step Five: Install NGINX

NGINX is an incredibly fast and light-weight web server. We will use it to serve up our static files for our Django app. To install it just run this command:

```
sudo apt-get install nginx
```

Keep in mind that you still need to start NGINX, but we will go over this in when we start configuring our server.

Step Six: Install Gunicorn

Gunicorn is a very powerful Python WSGI HTTP Server. Since it is a Python package we need to first activate our virtualenv to install it. Here is how we do that:

```
source /opt/myenv/bin/activate
```

Make sure you see the added "myenv" at the beginning of your terminal prompt. With your virtualenv now active, run this command:

```
pip install gunicorn
```

Gunicorn is now installed within your virtualenv.

If all you wanted was to get everything installed, feel free to stop here. Otherwise, please continue for instructions on how to configure everything to work together and make your app accessible to others on the web.

Step Seven: Configure PostgreSQL

Let's start off our configuration by working with PostgreSQL. With PostgreSQL we need to create a database, create a user, and grant the user we created access to the database we created. Start off by running the following command:

```
sudo su - postgres
```

Your terminal prompt should now say "postgres@yourserver". If this is the case, then run this command to create your database:

```
createdb mydb
```

Your database has now been created and is named "mydb" if you didn't change the command. You can name your database whatever you would like. Now create your database user with the following command:

```
createuser -P
```

You will now be met with a series of 6 prompts. The first one will ask you for the name of the new user. Use whatever name you would like. The next two prompts are for your password and confirmation of password for the new user. For the last 3 prompts just enter "n" and hit "enter". This just ensures your new users only has access to what you give it access to and nothing else. Now activate the PostgreSQL command line interface like so:

```
psql
```

Finally, grant this new user access to your new database with this command:

```
GRANT ALL PRIVILEGES ON DATABASE mydb TO myuser;
```

You now have a PostgreSQL database and a user to access that database with. Now we can install Django and set it up to use our new database.

Step Eight: Create a Django Project

In order to go any further we need a Django project to test with. This will allow us to see if what we are doing is working or not. Change directories into the directory of your virtualenv (in my case /opt/myenv) like so:

```
cd /opt/myenv
```

Now make sure your virtualenv is active. If you're unsure then just run the following command to ensure you're activated:

```
source /opt/myenv/bin/activate
```

With your virtualenv now active, run the following command to start a new Django project:

```
django-admin.py startproject myproject
```

You should see a new directory called "myproject" inside your virtualenv directory. This is where our new Django project files live.

In order for Django to be able to talk to our database we need to install a backend for PostgreSQL. Make sure your virtualenv is active and run the following command in order to do this:

```
pip install psycopg2
```

Change directories into the new "myproject" directory and then into it's subdirectory which is also called "myproject" like this:

```
cd /opt/myenv/myproject/myproject
```

Edit the settings.py file with your editor of choice:

```
nano settings.py
```

Find the database settings and edit them to look like this:

```
DATABASES = {
        'default': {
            'ENGINE': 'django.db.backends.postgresql_psycopg2', # Add
'postgresql_psycopg2', 'mysql', 'sqlite3' or 'oracle'.
            'NAME': 'mydb',                      # Or path to database file if
using sqlite3.
            # The following settings are not used with sqlite3:
            'USER': 'myuser',
            'PASSWORD': 'password',
            'HOST': 'localhost',                      # Empty for localhost
through domain sockets or           '127.0.0.1' for localhost through TCP.
            'PORT': '',                      # Set to empty string for default.
        }
    }
```

Save and exit the file. Now move up one directory so your in your main Django project directory (/opt/myenv/myproject).

```
cd /opt/myenv/myproject
```

Activate your virtualenv if you haven't already with the following command:

```
source /opt/myenv/bin/activate
```

With your virtualenv active, run the following command so that Django can add it's initial configuration and other tables to your database:

```
python manage.py syncdb
```

You should see some output describing what tables were installed, followed by a prompt asking if you want to create a superuser. This is optional and depends on if you will be using Django's auth system or the Django admin.

Step Nine: Configure Gunicorn

Gunicorn configuration is very specific to your applications needs. I will briefly go over running Gunicorn here with some different settings.

First lets just go over running Gunicorn with default settings. Here is the command to just run default Gunicorn:

```
gunicorn_django --bind yourdomainorip.com:8001
```

Be sure to replace "yourdomainorip.com" with your domain, or the IP address of your server if you prefer. Now go to your web browser and visit yourdomainorip.com:8001 and see what you get. You should get the Django welcome screen.

If you look closely at the output from the above command however, you will notice only one Gunicorn worker booted. What if you are launching a large-scale application on a large server? Have no fear! All we need to do is modify the command a bit like so:

```
gunicorn_django --workers=3 --bind yourdomainorip.com:8001
```

Now you will notice that 3 workers were booted instead of just 1 worker. You can change this number to whatever suits your needs.

Since we ran the command to start Gunicorn as root, Gunicorn is now running as root. What if you don't want that? Again, we can alter the command above slightly to accomodate:

```
gunicorn_django --workers=3 --user=nobody --bind yourdomainorip.com:8001
```

If you want to set more options for Gunicorn, then it is best to set up a config file that you can call when running Gunicorn. This will result in a much shorter and easier to read/configure Gunicorn command.

You can place the configuration file for gunicorn anywhere you would like. For simplicity, we will place it in our virtualenv directory. Navigate to the directory of your virtualenv like so:

```
cd /opt/myenv
```

Now open your config file with your preferred editor (nano is used in the example below):

```
sudo nano gunicorn_config.py
```

Add the following contents to the file:

```
command = '/opt/myenv/bin/gunicorn'
    pythonpath = '/opt/myenv/myproject'
    bind = '127.0.0.1:8001'
    workers = 3
    user = nobody
```

Save and exit the file. What these options do is to set the path to the gunicorn binary, add your project directory to your Python path, set the domain and port to bind Gunicorn to, set the number of gunicorn workers and set the user Gunicorn will run as.

In order to run the server, this time we need a bit longer command. Enter the following command into your prompt:

```
/opt/myenv/bin/gunicorn -c /opt/myenv/gunicorn_config.py myproject.wsgi
```

You will notice that in the above command we pass the "-c" flag. This tells gunicorn that we have a config file we want to use, which we pass in just after the "-c" flag. Lastly, we pass in a Python dotted notation reference to our WSGI file so that Gunicorn knows where our WSGI file is.

Running Gunicorn this way requires that you either run Gunicorn in its own screen session (if you're familiar with using screen), or that you background the process by hitting "ctrl + z" and then typing "bg" and "enter" all right after running the Gunicorn command. This will background the process so it continues running even after your current session is closed. This also poses the problem of needing to manually start or restart Gunicorn should your server gets rebooted or were it to crash for some reason. To solve this problem, most people use supervisord to manage Gunicorn and start/restart it as needed.

Lastly, this is by no means an exhaustive list of configuration options for Gunicorn. Please read the Gunicorn documentation found at gunicorn.org for more on this topic.

Step Ten: Configure NGINX

Before we get too carried away, let's first start NGINX like so:

```
sudo service nginx start
```

Since we are only setting NGINX to handle static files we need to first decide where our static files will be stored. Open your settings.py file for your Django project and edit the STATIC_ROOT line to look like this:

```
STATIC_ROOT = "/opt/myenv/static/"
```

This path can be wherever you would like. But for cleanliness, I typically put it just outside my Django project folder, but inside my virtualenv directory.

Now that you've set up where your static files will be located, let's configure NGINX to handle those files. Open up a new NGINX config file with the following command (you may replace "nano" with your editor of choice):

```
sudo nano /etc/nginx/sites-available/myproject
```

You can name the file whatever you would like, but the standard is typically to name it something related to the site that you are configuring. Now add the following to the file:

```
server {
        server_name yourdomainorip.com;
        access_log off;
        location /static/ {
            alias /opt/myenv/static/;
        }
        location / {
                proxy_pass http://127.0.0.1:8001;
                proxy_set_header X-Forwarded-Host $server_name;
                proxy_set_header X-Real-IP $remote_addr;
                add_header P3P 'CP="ALL DSP COR PSAa PSDa OUR NOR ONL UNI COM
NAV"';
        }
    }
```

Save and exit the file. The above configuration has set NGINX to serve anything requested at yourdomainorip.com/static/ from the static directory we set for our Django project. Anything requested at yourdomainorip.com will proxy to localhost on port 8001, which is where we told Gunicorn to run. The other lines ensure that the hostname and IP address of the request get passed on to Gunicorn. Without this, the IP address of every request becomes 127.0.0.1 and the hostname is just your server hostname.

Now we need to set up a symbolic link in the /etc/nginx/sites-enabled directory that points to this configuration file. That is how NGINX knows this site is active. Change directories to /etc/nginx/sites-enabled like this:

```
cd /etc/nginx/sites-enabled
```

Once there, run this command:

```
sudo ln -s ../sites-available/myproject
```

This will create the symbolic link we need so that NGINX knows to honor our new configuration file for our site.

Additionally, remove the default nginx server block:

`sudo rm default'

We need to restart NGINX though so that it knows to look for our changes. To do this run the following:

```
sudo service nginx restart
```

And that's it! You now have Django installed and working with PostgreSQL and your app is web accessible with NGINX serving static content and Gunicorn serving as your app server. If you have any questions or further advice, be sure to leave it in the comments section.