

By: Justin Ellingwood

 Subscribe

How To Set Up Django with Postgres, Nginx, and Gunicorn on Ubuntu 14.04

40

Posted Mar 18, 2015

 93.8k

Deployment

Django

Nginx

Python Frameworks

Python

Ubuntu

Introduction

Django is a powerful web framework that can help you get your Python application or website off the ground. Django includes a simplified development server for testing your code locally, but for anything even slightly production related, a more secure and powerful web server is required.

In this guide, we will demonstrate how to install and configure some components on Ubuntu 14.04 to support and serve Django applications. We will be setting up a PostgreSQL database instead of using the default SQLite database. We will configure the Gunicorn application server to interface with our applications. We will then set up Nginx to reverse proxy to Gunicorn, giving us access to its security and performance features to serve our apps.

Prerequisites and Goals

In order to complete this guide, you should have a fresh Ubuntu 14.04 server instance with a non-root user with `sudo` privileges configured. You can learn how to set this up by running through our [initial server setup guide](#).

We will be installing Django within a virtual environment. Installing Django into an environment specific to your project will allow your projects and their requirements to be handled separately.

Once we have our database and application up and running, we will install and configure the Gunicorn application server. This will serve as an interface to our application, translating client requests in HTTP to Python calls that our application can process. We will then set up Nginx in front of Gunicorn to take advantage of its high performance connection handling mechanisms and its easy-to-implement security features.

Let's get started.

Install the Packages from the Ubuntu Repositories

To begin the process, we'll download and install all of the items we need from the Ubuntu repositories. We will use the Python package manager `pip` to install additional components a bit later.

First, update the local package index and then download and install the packages:

```
sudo apt-get update
sudo apt-get install python-pip python-dev libpq-dev postgresql postgresql-contrib nginx
```

This will install `pip`, the Python development files needed to build Gunicorn later, the Postgres database system and the libraries needed to interact with it, and the Nginx web server.

Create the PostgreSQL Database and User

We're going to jump right in and create a database and database user for our Django application.

To work with Postgres in its default configuration, it is best to change to the `postgres` system user temporarily. Do that now by typing:

```
sudo su - postgres
```

When operating as the `postgres` user, you can log right into a PostgreSQL interactive session with no further authentication by typing:

```
psql
```

You will be given a PostgreSQL prompt where we can set up our requirements.

First, create a database for your project:

```
CREATE DATABASE myproject;
```

Every command must end with a semi-colon, so check that your command ends with one if you are experiencing issues.

Next, create a database user for our project. Make sure to select a secure password:

```
CREATE USER myprojectuser WITH PASSWORD 'password';
```

Now, we can give our new user access to administer our new database:

```
GRANT ALL PRIVILEGES ON DATABASE myproject TO myprojectuser;
```

When you are finished, exit out of the PostgreSQL prompt by typing:

```
\q
```

Now, exit out of the `postgres` user's shell session to get back to your normal user's shell session by typing:

```
exit
```

Create a Python Virtual Environment for your Project

Now that we have our database ready, we can begin getting the rest of our project requirements ready. We will be installing our Python requirements within a virtual environment for easier management.

To do this, we first need access to the `virtualenv` command. We can install this with `pip`:

```
sudo pip install virtualenv
```

With `virtualenv` installed, we can start forming our project. Create a directory where you wish to keep your project and move into the directory afterwards:

```
mkdir ~/myproject
cd ~/myproject
```

Within the project directory, create a Python virtual environment by typing:

```
virtualenv myprojectenv
```

This will create a directory called `myprojectenv` within your `myproject` directory. Inside, it will install a local version of Python and a local version of `pip`. We can use this to install and configure an isolated Python environment for our project.

Before we install our project's Python requirements, we need to activate the virtual environment. You can do that by typing:

```
source myprojectenv/bin/activate
```

Your prompt should change to indicate that you are now operating within a Python virtual environment. It will look something like this:

```
(myprojectenv)user@host:~/myproject$ .
```

With your virtual environment active, install Django, Gunicorn, and the `psycopg2` PostgreSQL adaptor with the local instance of `pip`:

```
pip install django gunicorn psycopg2
```

Create and Configure a New Django Project

With our Python components installed, we can create the actual Django project files.

Create the Django Project

Since we already have a project directory, we will tell Django to install the files here. It will create a second level directory with the actual code, which is normal, and place a management script in this directory. The key to this is the dot at the end that tells Django to create the files in the current directory:

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

Adjust the Project Settings

The first thing we should do with our newly created project files is adjust the settings. Open the settings file in your text editor:

```
nano myproject/settings.py
```

Start by finding the section that configures database access. It will start with `DATABASES`. The configuration in the file is for a SQLite database. We already created a PostgreSQL database for our project, so we need to adjust the settings.

Change the settings with your PostgreSQL database information. We tell Django to use the `psycopg2` adaptor we installed with `pip`. We need to give the database name, the database username, the database username's password, and then specify that the database is located on the local computer. You can leave the `PORT` setting as an empty string:

```
DATABASES = {
```

```
'default': {  
    'ENGINE': 'django.db.backends.postgresql_psycopg2',  
    'NAME': 'myproject',  
    'USER': 'myprojectuser',  
    'PASSWORD': 'password',  
    'HOST': 'localhost',  
    'PORT': '',  
}
```

Next, move down to the bottom of the file and add a setting indicating where the static files should be placed. This is necessary so that Nginx can handle requests for these items. The following line tells Django to place them in a directory called `static` in the base project directory:

```
STATIC_ROOT = os.path.join(BASE_DIR, "static/")
```

Save and close the file when you are finished.

Complete Initial Project Setup

Now, we can migrate the initial database schema to our PostgreSQL database using the management script:

```
cd ~/myproject  
./manage.py makemigrations  
./manage.py migrate
```

Create an administrative user for the project by typing:

```
./manage.py createsuperuser
```

You will have to select a username, provide an email address, and choose and confirm a password.

We can collect all of the static content into the directory location we configured by typing:

```
./manage.py collectstatic
```

You will have to confirm the operation. The static files will then be placed in a directory called `static` within your project directory.

Finally, you can test our your project by starting up the Django development server with this command:

```
./manage.py runserver 0.0.0.0:8000
```

In your web browser, visit your server's domain name or IP address followed by `:8000`:

```
http://server_domain_or_IP:8000
```

You should see the default Django index page:

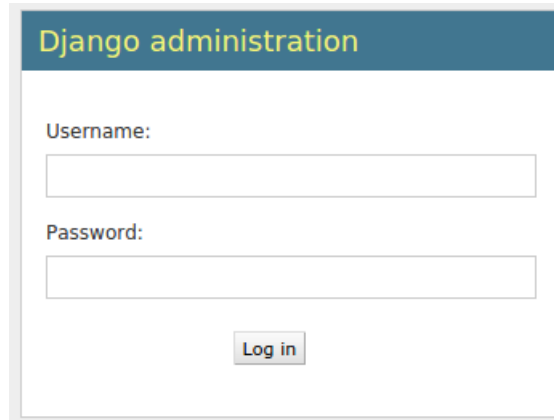
It worked!

Congratulations on your first Django-powered page.

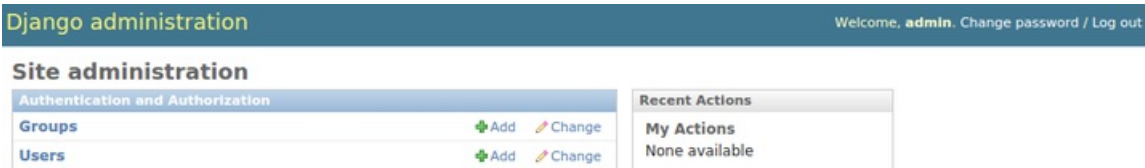
Of course, you haven't actually done any work yet. Next, start your first app by running `python manage.py startapp [app_label]`.

You're seeing this message because you have `DEBUG = True` in your Django settings file and you haven't configured any URLs. Get to work!

If you append `/admin` to the end of the URL in the address bar, you will be prompted for the administrative username and password you created with the `createsuperuser` command:

A screenshot of the Django administration login page. It has a blue header with the text "Django administration". Below the header, there are two input fields: "Username:" and "Password:". Below the password field is a "Log in" button.

After authenticating, you can access the default Django admin interface:

A screenshot of the Django administration interface after login. The header shows "Django administration" and "Welcome, admin. Change password / Log out". Below the header, there is a "Site administration" section with a table listing "Authentication and Authorization", "Groups", and "Users". Each row has "Add" and "Change" links. To the right, there is a "Recent Actions" section with a "My Actions" table showing "None available".

When you are finished exploring, hit CTRL-C in the terminal window to shut down the development server.

Testing Gunicorn's Ability to Serve the Project

The last thing we want to do before leaving our virtual environment is test Gunicorn to make sure that it can serve the application. We can do this easily by typing:

```
cd ~/myproject
gunicorn --bind 0.0.0.0:8000 myproject.wsgi:application
```

This will start Gunicorn on the same interface that the Django development server was running on. You can go back and test the app again. Note that the admin interface will not have any of the styling applied since Gunicorn does not know about the static content responsible for this.

We passed Gunicorn a module by specifying the relative directory path to Django's `wsgi.py` file, which is the entry point to our application, using Python's module syntax. Inside of this file, a function called `application` is defined, which is used to communicate with the application. To learn more about the WSGI specification, click [here](#).

When you are finished testing, hit CTRL-C in the terminal window to stop Gunicorn.

We're now finished configuring our Django application. We can back out of our virtual environment by typing:

Create a Gunicorn Upstart File

We have tested that Gunicorn can interact with our Django application, but we should implement a more robust way of starting and stopping the application server. To accomplish this, we'll make an Upstart script.

Create and open an Upstart file for Gunicorn with `sudo` privileges in your text editor:

```
sudo nano /etc/init/gunicorn.conf
```

We'll start with a simple description string to state what our service file is for. We'll then move on to defining the system runlevels where this service should be automatically started. The normal runlevels to run services are 2, 3, 4, and 5. We'll run our service when the system is in any of those. We'll tell it to stop when its in any other runlevel (such as when the system is rebooting, shutting down, or in single-user mode):

```
description "Gunicorn application server handling myproject"

start on runlevel [2345]
stop on runlevel [!2345]
```

Next, we'll tell Upstart to automatically restart the service if it fails. We also want to specify the user and group to run under. We'll use our normal user since all of our files are owned by that user. We'll let the `www-data` group which Nginx belongs to be the group owners. We also need to change to our project's directory so that the Gunicorn commands execute correctly:

```
description "Gunicorn application server handling myproject"

start on runlevel [2345]
stop on runlevel [!2345]

respawn
setuid user
setgid www-data
chdir /home/user/myproject
```

Now, we just need to give the command that will start the Gunicorn process. We need to give the path to the Gunicorn executable, which is stored within our virtual environment. We will tell it to use a Unix socket instead of a network port to communicate with Nginx, since both services will be running on this server. This is more secure and faster. You can add any other configuration for Gunicorn here as well. For instance, we'll specify that we want 3 worker processes:

```
description "Gunicorn application server handling myproject"

start on runlevel [2345]
stop on runlevel [!2345]

respawn
setuid user
setgid www-data
chdir /home/user/myproject

exec myprojectenv/bin/gunicorn --workers 3 --bind unix:/home/user/myproject/myproject.sock myproject.wsgi:application
```

When you are finished, save and close the file.

Start the Gunicorn service by typing:

```
sudo service gunicorn start
```

Configure Nginx to Proxy Pass to Gunicorn

Now that Gunicorn is set up, we need to configure Nginx to pass traffic to the process.

Start by creating and opening a new server block in Nginx's `sites-available` directory:

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

Inside, open up a new server block. We will start by specifying that this block should listen on the normal port 80 and that it should respond to our server's domain name or IP address:

```
server {  
    listen 80;  
    server_name server_domain_or_IP;  
}
```

Next, we will tell Nginx to ignore any problems with finding a favicon. We will also tell it where to find the static assets that we collected in our `~/myproject/static` directory. All of these files have a standard URI prefix of `"/static"`, so we can create a location block to match those requests:

```
server {  
    listen 80;  
    server_name server_domain_or_IP;  
  
    location = /favicon.ico { access_log off; log_not_found off; }  
    location /static/ {  
        root /home/user/myproject;  
    }  
}
```

Finally, we'll create a `location / { }` block to match all other requests. Inside of this location, we'll include the standard `proxy_params` file included with the Nginx installation and then we will pass the traffic to the socket that our Gunicorn process created:

```
server {  
    listen 80;  
    server_name server_domain_or_IP;  
  
    location = /favicon.ico { access_log off; log_not_found off; }  
    location /static/ {  
        root /home/user/myproject;  
    }  
  
    location / {  
        include proxy_params;  
        proxy_pass http://unix:/home/user/myproject/myproject.sock;
```

```
}  
}
```

Save and close the file when you are finished. Now, we can enable the file by linking it to the `sites-enabled` directory:

```
sudo ln -s /etc/nginx/sites-available/myproject /etc/nginx/sites-enabled
```

Test your Nginx configuration for syntax errors by typing:

```
sudo nginx -t
```

If no errors are reported, go ahead and restart Nginx by typing:

```
sudo service nginx restart
```

You should now be able to go to your server's domain or IP address to view your application.

Conclusion

In this guide, we've set up a Django project in its own virtual environment. We've configured Gunicorn to translate client requests so that Django can handle them. Afterwards, we set up Nginx to act as a reverse proxy to handle client connections and serve the correct project depending on the client request.

Django makes creating projects and applications simple by providing many of the common pieces, allowing you to focus on the unique elements. By leveraging the general tool chain described in this article, you can easily serve the applications you create from a single server.

By: Justin Ellingwood

Upvote (40)

 Subscribe



Author:
Justin Ellingwood

Spin up an SSD cloud server in under a minute.

Simple setup. Full root access.
Straightforward pricing.

DEPLOY SERVER



Related Tutorials

[How to Deploy a Node.js App Using Terraform on Ubuntu 14.04](#)

[How To Set Up Django with Postgres, Nginx, and Gunicorn on Ubuntu 16.04](#)

[How To Use PostgreSQL with your Django Application on Ubuntu 16.04](#)

[How To Serve Django Applications with Apache and mod_wsgi on Ubuntu 16.04](#)

[How To Serve Django Applications with uWSGI and Nginx on Ubuntu 16.04](#)

82 Comments

Leave a comment...


[Log In to Comment](#)



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.



Copyright © 2016 DigitalOcean™ Inc.

[Community](#) [Tutorials](#) [Questions](#) [Projects](#) [Tags](#) [Newsletter](#) [RSS](#) 

[Distros & One-Click Apps](#) [Terms, Privacy, & Copyright](#) [Security](#) [Report a Bug](#) [Get Paid to Write](#)