DigitalOcean

Subscribe   Share   Contents

How To Set Up Django with Postgres, Nginx, and Gunicorn on Debian 9

2

Posted September 6, 2018   5.8k   DJANGO   PYTHON   PYTHON FRAMEWORKS   DATABASES   POSTGRESQL   NGINX   DEBIAN 9

By: Justin Ellingwood   By: Hanif Jetha

Not using **Debian 9**? Choose a different version:

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 Deb
and serve Django applications. We will be setting up a PostgreSQL database instead of us

SCROLL TO TOP

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

In order to complete this guide, you should have a fresh Debian 9 server instance with a basic firewall and 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 from 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.

Step 1 — Installing the Packages from the Debian Repositories

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

We need to update the local `apt` package index and then download and install the packages. The packages we install depend on which version of Python your project will use.

If you are using Django with **Python 3**, type:

```
$ sudo apt update
$ sudo apt install python3-pip python3-dev libpq-dev postgresql postgresql-contrib nginx curl
```

Django 1.11 is the last release of Django that will support Python 2. If you are starting new projects, it is strongly recommended that you choose Python 3. If you still need to use **Python 2**, type:

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

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.

Step 2 — Creating the PostgreSQL Database and Us

SCROLL TO TOP

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

By default, Postgres uses an authentication scheme called "peer authentication" for local connections. Basically, this means that if the user's operating system username matches a valid Postgres username, that user can login with no further authentication.

During the Postgres installation, an operating system user named `postgres` was created to correspond to the `postgres` PostgreSQL administrative user. We need to use this user to perform administrative tasks. We can use sudo and pass in the username with the `-u` option.

Log into an interactive Postgres session by typing:

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

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

First, create a database for your project:

```
postgres=# CREATE DATABASE myproject;
```

> **Note:** Every Postgres statement must end with a semi-colon, so make sure 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:

```
postgres=# CREATE USER myprojectuser WITH PASSWORD 'password';
```

Afterwards, we'll modify a few of the connection parameters for the user we just created. This will speed up database operations so that the correct values do not have to be queried and set each time a connection is established.

We are setting the default encoding to `UTF-8`, which Django expects. We are also setting the default transaction isolation scheme to "read committed", which blocks reads from uncommitted transactions. Lastly, we are setting the timezone. By default, our Django projects will be set to use `UTC`. These are all recommendations from the Django project itself:

```
postgres=# ALTER ROLE myprojectuser SET client_encoding TO 'utf8';
postgres=# ALTER ROLE myprojectuser SET default_transaction_isolation TO 'read committed';
postgres=# ALTER ROLE myprojectuser SET timezone TO 'UTC';
```

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

```
postgres=# GRANT ALL PRIVILEGES ON DATABASE myproject TO myprojectuser;
```

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

```
postgres=# \q
```

Postgres is now set up so that Django can connect to and manage its database information.

Step 3 — Creating a Python Virtual Environment for your Project

Now that we have our database, 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`.

If you are using **Python 3**, upgrade `pip` and install the package by typing:

```
$ sudo -H pip3 install --upgrade pip
$ sudo -H pip3 install virtualenv
```

If you are using **Python 2**, upgrade `pip` and install the package by typing:

```
$ sudo -H pip install --upgrade pip
$ sudo -H pip install virtualenv
```

With `virtualenv` installed, we can start forming our project. Create and move into a directory where we can keep our project files:

```
$ mkdir ~/myprojectdir
$ cd ~/myprojectdir
```

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

```
$ virtualenv myprojectenv
```

This will create a directory called `myprojectenv` within your `myprojectdir` 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:

SCROLL TO TOP

```
$ 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:~/myprojectdir$`.

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

> **Note:** When the virtual environment is activated (when your prompt has `(myprojectenv)` preceding it), use `pip` instead of `pip3`, even if you are using Python 3. The virtual environment's copy of the tool is always named `pip`, regardless of the Python version.

```
(myprojectenv) $ pip install django gunicorn psycopg2-binary
```

You should now have all of the software needed to start a Django project.

Step 4 — Creating and Configuring a New Django Project

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

Creating 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 that we are defining the directory explicitly instead of allowing Django to make decisions relative to our current directory:

```
(myprojectenv) $ django-admin.py startproject myproject ~/myprojectdir
```

At this point, your project directory (`~/myprojectdir` in our case) should have the following content:

- `~/myprojectdir/manage.py`: A Django project management script.
- `~/myprojectdir/myproject/`: The Django project package. This should contain the `__init__.py`, `settings.py`, `urls.py`, and `wsgi.py` files.
- `~/myprojectdir/myprojectenv/`: The virtual environment directory we created earlier.

Adjusting 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:

SCROLL TO TOP

```
(myprojectenv) $ nano ~/myprojectdir/myproject/settings.py
```

Start by locating the `ALLOWED_HOSTS` directive. This defines a list of the server's addresses or domain names may be used to connect to the Django instance. Any incoming requests with a **Host** header that is not in this list will raise an exception. Django requires that you set this to prevent a certain class of security vulnerability.

In the square brackets, list the IP addresses or domain names that are associated with your Django server. Each item should be listed in quotations with entries separated by a comma. If you wish requests for an entire domain and any subdomains, prepend a period to the beginning of the entry. In the snippet below, there are a few commented out examples used to demonstrate:

> **Note:** Be sure to include `localhost` as one of the options since we will be proxying connections through a local Nginx instance.

~/myprojectdir/myproject/settings.py

```
. . .
# The simplest case: just add the domain name(s) and IP addresses of your Django server
# ALLOWED_HOSTS = [ 'example.com', '203.0.113.5']
# To respond to 'example.com' and any subdomains, start the domain with a dot
# ALLOWED_HOSTS = ['.example.com', '203.0.113.5']
ALLOWED_HOSTS = ['your_server_domain_or_IP', 'second_domain_or_IP', . . ., 'localhost']
```

Next, find 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 user's password, and then specify that the database is located on the local computer. You can leave the `PORT` setting as an empty string:

~/myprojectdir/myproject/settings.py

```
. . .

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:

~/myprojectdir/myproject/settings.py

```
. . .

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

Save and close the file when you are finished.

Completing Initial Project Setup

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

```
(myprojectenv) $ ~/myprojectdir/manage.py makemigrations
(myprojectenv) $ ~/myprojectdir/manage.py migrate
```

Create an administrative user for the project by typing:

```
(myprojectenv) $ ~/myprojectdir/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:

```
(myprojectenv) $ ~/myprojectdir/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.

If you followed the initial server setup guide, you should have a UFW firewall protecting your server. In order to test the development server, we'll have to allow access to the port we'll be using.

Create an exception for port 8000 by typing:

```
(myprojectenv) $ sudo ufw allow 8000
```

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

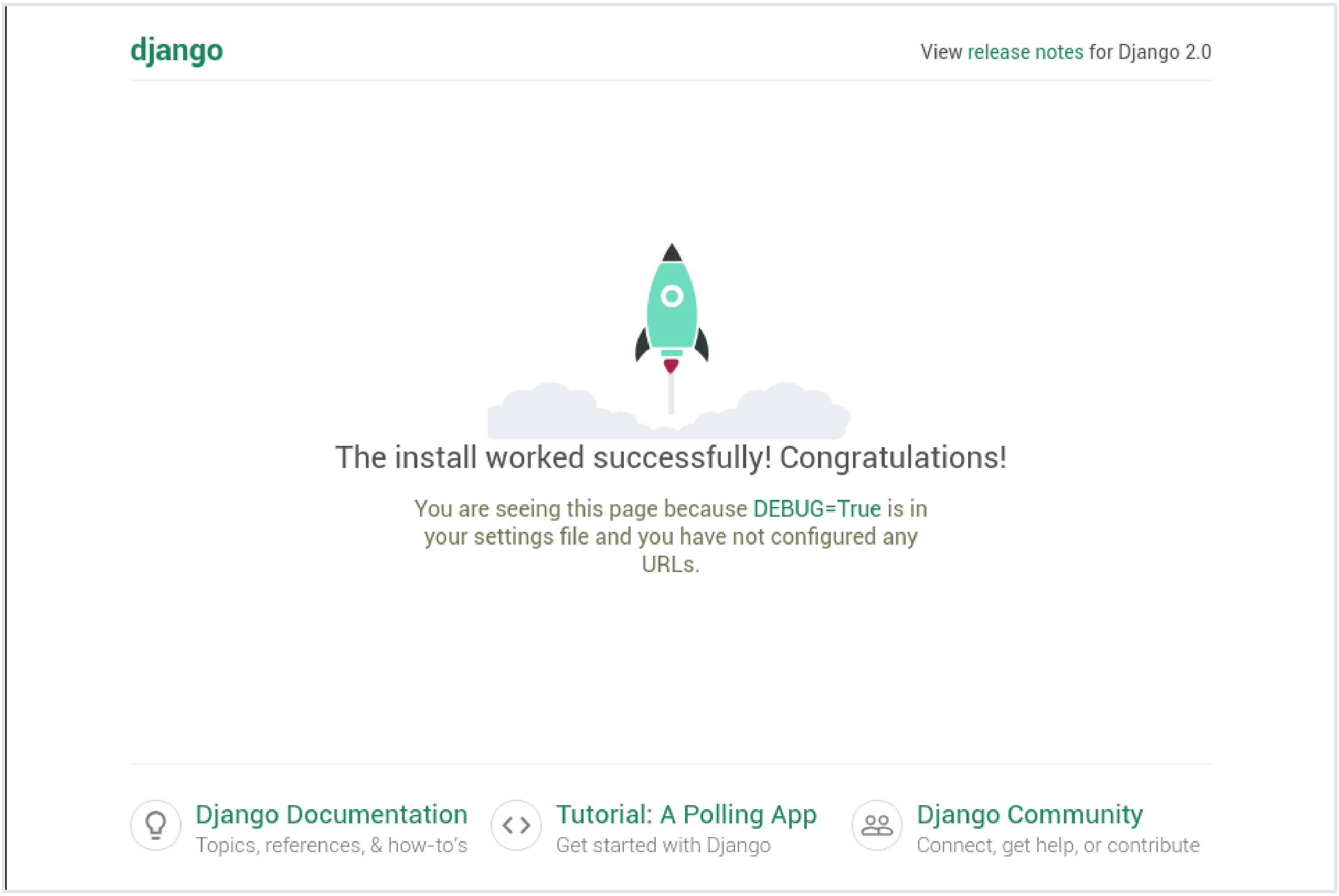
SCROLL TO TOP

```
(myprojectenv) $ ~/myprojectdir/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:


django

View release notes for Django 2.0

The install worked successfully! Congratulations!

You are seeing this page because DEBUG=True is in your settings file and you have not configured any URLs.

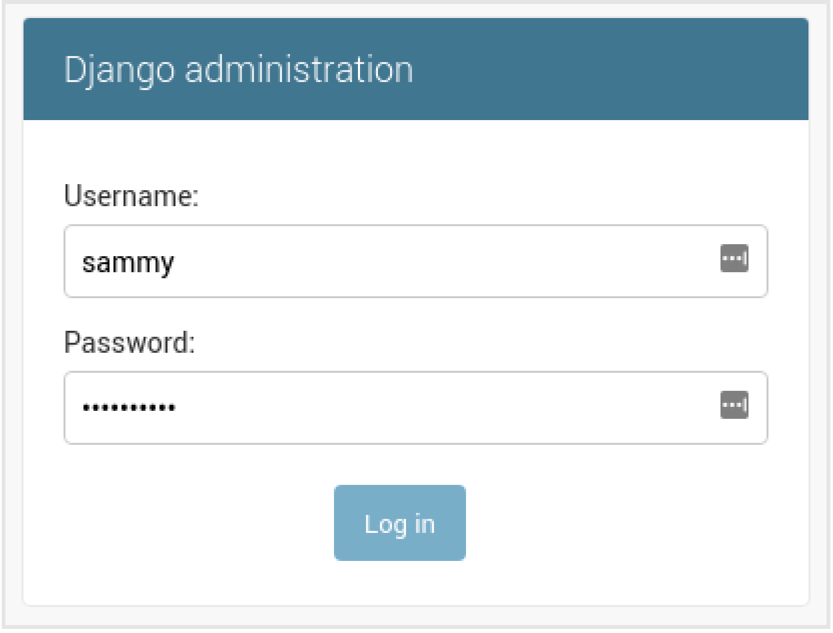
Django Documentation
Topics, references, & how-to's

Tutorial: A Polling App
Get started with Django

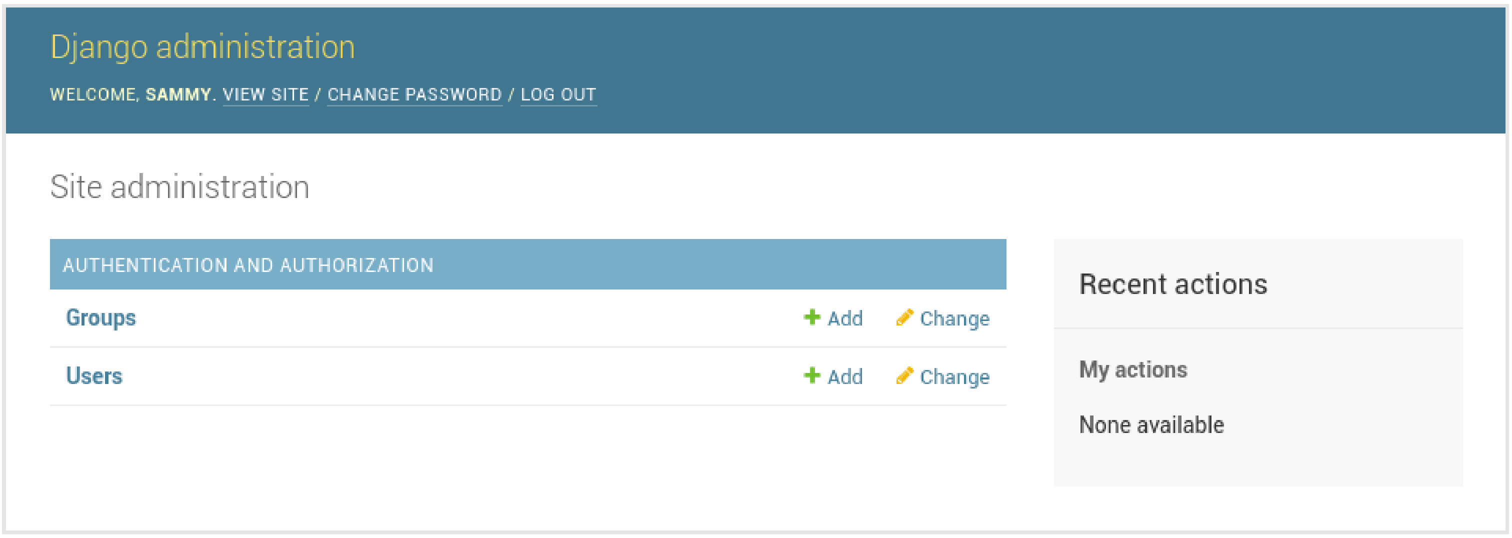
Django Community
Connect, get help, or contribute

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:

SCROLL TO TOP



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



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 by entering our project directory and using `gunicorn` to load the project's WSGI module:

```
(myprojectenv) $ cd ~/myprojectdir
(myprojectenv) $ gunicorn --bind 0.0.0.0:8000 myproject.wsgi
```

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: The admin interface will not have any of the styling applied since Gunicorn does not know how to find the static CSS content responsible for this.

SCROLL TO TOP

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:

```
(myprojectenv) $ deactivate
```

The virtual environment indicator in your prompt will be removed.

Step 5 — Creating systemd Socket and Service Files for Gunicorn

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 systemd service and socket files.

The Gunicorn socket will be created at boot and will listen for connections. When a connection occurs, systemd will automatically start the Gunicorn process to handle the connection.

Start by creating and opening a systemd socket file for Gunicorn with `sudo` privileges:

```
$ sudo nano /etc/systemd/system/gunicorn.socket
```

Inside, we will create a `[Unit]` section to describe the socket, a `[Socket]` section to define the socket location, and an `[Install]` section to make sure the socket is created at the right time:

/etc/systemd/system/gunicorn.socket

```
[Unit]
Description=gunicorn socket

[Socket]
ListenStream=/run/gunicorn.sock

[Install]
WantedBy=sockets.target
```

Save and close the file when you are finished.

SCROLL TO TOP

Next, create and open a systemd service file for Gunicorn with `sudo` privileges in your text editor. The service filename should match the socket filename with the exception of the extension:

```
$ sudo nano /etc/systemd/system/gunicorn.service
```

Start with the `[Unit]` section, which is used to specify metadata and dependencies. We'll put a description of our service here and tell the init system to only start this after the networking target has been reached. Because our service relies on the socket from the socket file, we need to include a `Requires` directive to indicate that relationship:

/etc/systemd/system/gunicorn.service

```
[Unit]
Description=gunicorn daemon
Requires=gunicorn.socket
After=network.target
```

Next, we'll open up the `[Service]` section. We'll specify the user and group that we want to process to run under. We will give our regular user account ownership of the process since it owns all of the relevant files. We'll give group ownership to the `www-data` group so that Nginx can communicate easily with Gunicorn.

We'll then map out the working directory and specify the command to use to start the service. In this case, we'll have to specify the full path to the Gunicorn executable, which is installed within our virtual environment. We will bind the process to the Unix socket we created within the `/run` directory so that the process can communicate with Nginx. We log all data to standard output so that the `journald` process can collect the Gunicorn logs. We can also specify any optional Gunicorn tweaks here. For example, we specified 3 worker processes in this case:

/etc/systemd/system/gunicorn.service

```
[Unit]
Description=gunicorn daemon
Requires=gunicorn.socket
After=network.target

[Service]
User=sammy
Group=www-data
WorkingDirectory=/home/sammy/myprojectdir
ExecStart=/home/sammy/myprojectdir/myprojectenv/bin/gunicorn \
          --access-logfile - \
          --workers 3 \
          --bind unix:/run/gunicorn.sock \
          myproject.wsgi:application
```

Finally, we'll add an `[Install]` section. This will tell systemd what to link this service to if we enable it to start at boot. We want this service to start when the regular multi-user system is up and r

SCROLL TO TOP

/etc/systemd/system/gunicorn.service

```
[Unit]
Description=gunicorn daemon
Requires=gunicorn.socket
After=network.target

[Service]
User=sammy
Group=www-data
WorkingDirectory=/home/sammy/myprojectdir
ExecStart=/home/sammy/myprojectdir/myprojectenv/bin/gunicorn \
          --access-logfile - \
          --workers 3 \
          --bind unix:/run/gunicorn.sock \
          myproject.wsgi:application

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

With that, our systemd service file is complete. Save and close it now.

We can now start and enable the Gunicorn socket. This will create the socket file at `/run/gunicorn.sock` now and at boot. When a connection is made to that socket, systemd will automatically start the `gunicorn.service` to handle it:

```
$ sudo systemctl start gunicorn.socket
$ sudo systemctl enable gunicorn.socket
```

We can confirm that the operation was successful by checking for the socket file.

Step 6 — Checking for the Gunicorn Socket File

Check the status of the process to find out whether it was able to start:

```
$ sudo systemctl status gunicorn.socket
```

Next, check for the existence of the `gunicorn.sock` file within the `/run` directory:

```
$ file /run/gunicorn.sock
```

```
Output
/run/gunicorn.sock: socket
```

SCROLL TO TOP

If the `systemctl status` command indicated that an error occurred or if you do not find the `gunicorn.sock` file in the directory, it's an indication that the Gunicorn socket was not able to be created correctly. Check the Gunicorn socket's logs by typing:

```
$ sudo journalctl -u gunicorn.socket
```

Take another look at your `/etc/systemd/system/gunicorn.socket` file to fix any problems before continuing.

Step 7 — Testing Socket Activation

Currently, if you've only started the `gunicorn.socket` unit, the `gunicorn.service` will not be active yet since the socket has not yet received any connections. You can check this by typing:

```
$ sudo systemctl status gunicorn
```

```
Output

● gunicorn.service - gunicorn daemon
   Loaded: loaded (/etc/systemd/system/gunicorn.service; disabled; vendor preset: enabled)
   Active: inactive (dead)
```

To test the socket activation mechanism, we can send a connection to the socket through `curl` by typing:

```
$ curl --unix-socket /run/gunicorn.sock localhost
```

You should see the HTML output from your application in the terminal. This indicates that Gunicorn was started and was able to serve your Django application. You can verify that the Gunicorn service is running by typing:

```
$ sudo systemctl status gunicorn
```

```
Output
● gunicorn.service - gunicorn daemon
   Loaded: loaded (/etc/systemd/system/gunicorn.service; disabled; vendor preset: enabled)
   Active: active (running) since Mon 2018-07-09 20:00:40 UTC; 4s ago
 Main PID: 1157 (gunicorn)
    Tasks: 4 (limit: 1153)
   CGroup: /system.slice/gunicorn.service
           ├─1157 /home/sammy/myprojectdir/myprojectenv/bin/python3 /home/sammy/myprojectdir/myprojec
           ├─1178 /home/sammy/myprojectdir/myprojectenv/bin/python3 /home/sammy/myprojectdir/myprojec
           ├─1180 /home/sammy/myprojectdir/myprojectenv/bin/python3 /home/sammy/[illegible]jec
           └─1181 /home/sammy/myprojectdir/myprojectenv/bin/python3 /home/sammy/myprojectdir/myprojec

Jul 09 20:00:40 django1 systemd[1]: Started gunicorn daemon.
Jul 09 20:00:40 django1 gunicorn[1157]: [2018-07-09 20:00:40 +0000] [1157] [INFO] Starting gunicorn 1
Jul 09 20:00:40 django1 gunicorn[1157]: [2018-07-09 20:00:40 +0000] [1157] [INFO] Listening at: unix:
Jul 09 20:00:40 django1 gunicorn[1157]: [2018-07-09 20:00:40 +0000] [1157] [INFO] Using worker: sync
Jul 09 20:00:40 django1 gunicorn[1157]: [2018-07-09 20:00:40 +0000] [1178] [INFO] Booting worker with
Jul 09 20:00:40 django1 gunicorn[1157]: [2018-07-09 20:00:40 +0000] [1180] [INFO] Booting worker with
Jul 09 20:00:40 django1 gunicorn[1157]: [2018-07-09 20:00:40 +0000] [1181] [INFO] Booting worker with
Jul 09 20:00:41 django1 gunicorn[1157]:  - - [09/Jul/2018:20:00:41 +0000] "GET / HTTP/1.1" 200 16348
```

If the output from `curl` or the output of `systemctl status` indicates that a problem occurred, check the logs for additional details:

```
$ sudo journalctl -u gunicorn
```

Check your `/etc/systemd/system/gunicorn.service` file for problems. If you make changes to the `/etc/systemd/system/gunicorn.service` file, reload the daemon to reread the service definition and restart the Gunicorn process by typing:

```
$ sudo systemctl daemon-reload
$ sudo systemctl restart gunicorn
```

Make sure you troubleshoot the above issues before continuing.

Step 8 — 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:

/etc/nginx/sites-available/myproject

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

SCROLL TO TOP

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 `~/myprojectdir/static` directory. All of these files have a standard URI prefix of "/static", so we can create a location block to match those requests:

/etc/nginx/sites-available/myproject

```
server {
    listen 80;
    server_name server_domain_or_IP;

    location = /favicon.ico { access_log off; log_not_found off; }
    location /static/ {
        root /home/sammy/myprojectdir;
    }
}
```

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 directly to the Gunicorn socket:

/etc/nginx/sites-available/myproject

```
server {
    listen 80;
    server_name server_domain_or_IP;

    location = /favicon.ico { access_log off; log_not_found off; }
    location /static/ {
        root /home/sammy/myprojectdir;
    }

    location / {
        include proxy_params;
        proxy_pass http://unix:/run/gunicorn.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
```

SCROLL TO TOP

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

```
$ sudo systemctl restart nginx
```

Finally, we need to open up our firewall to normal traffic on port 80. Since we no longer need access to the development server, we can remove the rule to open port 8000 as well:

```
$ sudo ufw delete allow 8000
$ sudo ufw allow 'Nginx Full'
```

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

Note: After configuring Nginx, the next step should be securing traffic to the server using SSL/TLS. This is important because without it, all information, including passwords are sent over the network in plain text.

If you have a domain name, the easiest way get an SSL certificate to secure your traffic is using Let's Encrypt. Follow this guide to set up Let's Encrypt with Nginx on Debian 9. Follow the procedure using the Nginx server block we created in this guide.

If you do not have a domain name, you can still secure your site for testing and learning with a self-signed SSL certificate. Again, follow the process using the Nginx server block we created in this tutorial.

Troubleshooting Nginx and Gunicorn

If this last step does not show your application, you will need to troubleshoot your installation.

Nginx Is Showing the Default Page Instead of the Django Application

If Nginx displays the default page instead of proxying to your application, it usually means that you need to adjust the `server_name` within the `/etc/nginx/sites-available/myproject` file to point to your server's IP address or domain name.

Nginx uses the `server_name` to determine which server block to use to respond to requests. If you are seeing the default Nginx page, it is a sign that Nginx wasn't able to match the request to a sever block explicitly, so it's falling back on the default block defined in `/etc/nginx/sites-available/default`.

The `server_name` in your project's server block must be more specific than the one in the default server block to be selected.

Nginx Is Displaying a 502 Bad Gateway Error Instead of the Django Application

A 502 error indicates that Nginx is unable to successfully proxy the request. A wide range of configuration problems express themselves with a 502 error, so more information is required to troubl

SCROLL TO TOP

The primary place to look for more information is in Nginx's error logs. Generally, this will tell you what conditions caused problems during the proxying event. Follow the Nginx error logs by typing:

```
$ sudo tail -F /var/log/nginx/error.log
```

Now, make another request in your browser to generate a fresh error (try refreshing the page). You should see a fresh error message written to the log. If you look at the message, it should help you narrow down the problem.

You might see some of the following message:

connect() to unix:/run/gunicorn.sock failed (2: No such file or directory)

This indicates that Nginx was unable to find the `gunicorn.sock` file at the given location. You should compare the `proxy_pass` location defined within `/etc/nginx/sites-available/myproject` file to the actual location of the `gunicorn.sock` file generated by the `gunicorn.socket` systemd unit.

If you cannot find a `gunicorn.sock` file within the `/run` directory, it generally means that the systemd socket file was unable to create it. Go back to the section on checking for the Gunicorn socket file to step through the troubleshooting steps for Gunicorn.

connect() to unix:/run/gunicorn.sock failed (13: Permission denied)

This indicates that Nginx was unable to connect to the Gunicorn socket because of permissions problems. This can happen when the procedure is followed using the root user instead of a `sudo` user. While systemd is able to create the Gunicorn socket file, Nginx is unable to access it.

This can happen if there are limited permissions at any point between the root directory (`/`) the `gunicorn.sock` file. We can see the permissions and ownership values of the socket file and each of its parent directories by passing the absolute path to our socket file to the `namei` command:

```
$ namei -l /run/gunicorn.sock
```

```
Output
f: /run/gunicorn.sock
drwxr-xr-x root root /
drwxr-xr-x root root run
srw-rw-rw- root root gunicorn.sock
```

The output displays the permissions of each of the directory components. By looking at the permissions (first column), owner (second column) and group owner (third column), we can figure out what type of access is allowed to the socket file.

In the above example, the socket file and each of the directories leading up to the socke SCROLL TO TOP read and execute permissions (the permissions column for the directories end with `r-x` instead of `---`).

The Nginx process should be able to access the socket successfully.

If any of the directories leading up to the socket do not have world read and execute permission, Nginx will not be able to access the socket without allowing world read and execute permissions or making sure group ownership is given to a group that Nginx is a part of.

Django Is Displaying: "could not connect to server: Connection refused"

One message that you may see from Django when attempting to access parts of the application in the web browser is:

```
OperationalError at /admin/login/
could not connect to server: Connection refused
    Is the server running on host "localhost" (127.0.0.1) and accepting
    TCP/IP connections on port 5432?
```

This indicates that Django is unable to connect to the Postgres database. Make sure that the Postgres instance is running by typing:

```
$ sudo systemctl status postgresql
```

If it is not, you can start it and enable it to start automatically at boot (if it is not already configured to do so) by typing:

```
$ sudo systemctl start postgresql
$ sudo systemctl enable postgresql
```

If you are still having issues, make sure the database settings defined in the `~/myprojectdir/myproject/settings.py` file are correct.

Further Troubleshooting

For additional troubleshooting, the logs can help narrow down root causes. Check each of them in turn and look for messages indicating problem areas.

The following logs may be helpful:

- Check the Nginx process logs by typing: `sudo journalctl -u nginx`
- Check the Nginx access logs by typing: `sudo less /var/log/nginx/access.log`
- Check the Nginx error logs by typing: `sudo less /var/log/nginx/error.log`
- Check the Gunicorn application logs by typing: `sudo journalctl -u gunicorn`
- Check the Gunicorn socket logs by typing: `sudo journalctl -u gunicorn.socket`

SCROLL TO TOP

As you update your configuration or application, you will likely need to restart the processes to adjust to your changes.

If you update your Django application, you can restart the Gunicorn process to pick up the changes by typing:

```
$ sudo systemctl restart gunicorn
```

If you change Gunicorn socket or service files, reload the daemon and restart the process by typing:

```
$ sudo systemctl daemon-reload
$ sudo systemctl restart gunicorn.socket gunicorn.service
```

If you change the Nginx server block configuration, test the configuration and then Nginx by typing:

```
$ sudo nginx -t && sudo systemctl restart nginx
```

These commands are helpful for picking up changes as you adjust your configuration.

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 By: Hanif Jetha

Related Tutorials

Bias-Variance for Deep Reinforcement Learning: How To Build a Bot for Atari with OpenAI Gym

How To Troubleshoot Issues in MySQL

How To Set Up Jupyter Notebook with Python 3 on Ubuntu 18.04

How To Set Up a Remote Database to Optimize Site Performance with MySQL on Ubuntu 18.04

How To Send Web Push Notifications from Django Applications

2 Comments

Leave a comment...

Log In to Comment

Vyachez *November 16, 2018*

1 Thank you! This was very helpful - finally made my Postgres working there!

SCROLL TO TOP

0 bukosabino *January 6, 2019*

Hey, thank you! Really good tutorial!

I have my django app working on Debian 9 using nginx and gunicorn.

But, I have a problem with the static files. Could be a permission (chown/chmod) problem?