**The Process Manager: Green Unicorn**

In general, the process manager performs the following roles:

1. Restart the app automatically if it crashes.
2. Gain insights into runtime performance and resource consumption.
3. Modify settings dynamically to improve performance.
4. Load Balancing

Green Unicorn, AKA Gunicorn, performs these roles by acting as an intermediary between incoming requests and your Django app. While you were testing your app on your local machine, it was ok to use Django’s built-in, light-weight testing server, but that server wasn’t designed to be used in a deployment environment.

Gunicorn, on the other hand, is designed to handle incoming requests that will need to be routed to the interpreter. This includes any route you’ve added to urls.py. We’ll learn more about our other server, Nginx, which will be serving static files such as images.

Gunicorn needs a way to communicate with our app. When creating a new Django project you may have noticed that a document called wsgi.py is automatically generated. What you need to know is that your wsgi.py file is the glue that connects the Gunicorn server to your Django app. It’s how Gunicorn knows where to look for all that good code you wrote!

**Note:** How WSGI works, what it is, and why it’s needed is a bit outside our scope here. In order to truly understand what WSGI does, you’ll need to dig into some more complex topics. To try to explain it briefly here would be a disservice. As always, when you read about or hear of something new, be curious. Go out there and learn more. A good place to start is by reading the [Python docs on the topic](https://www.python.org/dev/peps/pep-3333/).

Chances are that when you deploy an app you’ve been developing in a local environment, you need to use some kind of process manager, regardless of the technology stack. Some common combinations include Unicorn for Rails, pm2 for NodeJS, and Apache for PHP.

There are three basic functions you will need to set up when you add Gunicorn to your project. To add this functionality, Gunicorn needs some instructions on how to do the following:

* Which socket to connect to
* What to do if a process fails
* How many workers to set up

Armed with these instructions, Gunicorn takes over the job of ensuring that your app keeps running if an error occurs, directing traffic to the appropriate port, and running concurrent processes. In short, Gunicorn makes sure your deployment server runs more smoothly.

**Setup**

Before we create our server we have to do some setup first. Let's get started!

Today you will be deploying your Courses app from the Django models chapter. When you are finished, your virtual machine will be set up to serve any Django project. This will enable you to make an easy swap with a new project, for example, your belt exam. We'll show you how later.

In this first section, you will be navigating around on your local machine. If you are using Unix (Mac), or Linux, these commands are for you. If you are on Windows, you will want to use git bash in order for these commands to run properly.

**Step 1: Getting Started**

Get started by activating your project's virtual environment.

Once you have activated your virtual environment, cd into your project directory. You will now save all of your installed pip modules into a .txt file. Later, we will use this file to install all of the required pip modules on our remote machine with a single command.

pip freeze > requirements.txt

**In your text editor, open your requirements.txt file and, if they exist, remove pygraphviz, pydot and anything with MySQL in it. These modules can be tricky to install and require additional installations, so we remove them now to prevent problems later.**

**Step 2: Committing**

**Important!**

We’re about to initialize a new git repo. Your git repo must be initialized within the outer project folder. This is the same level as your manage.py file. If you ls and don’t see manage.py, you are in the wrong place. Double check your location before you initialize your repo.

First we’ll create a .gitignore file.

touch .gitignore

As the name implies, your gitignore file tells git to ignore any files, directories, etc. you include in the file. In this case, we’re instructing git to ignore all files with the extension “pyc”, as well as your virtual environment, if you have placed one inside of your project folder, in your gitignore file, venv/ should be replaced with the name of your virtual environment.

Open your .gitignore file in your text editor and add the lines:

\*.pyc

venv/

We know this is familiar, but here’s a reminder of how to initialize a new repo:

> git init

> git add --all

> git commit -m "initial commit"

Create a new github repo and, back in terminal run these commands, replacing the repo url with your own.

> git remote add origin https://github.com/AnnaBNana/django\_courses.git

> git push origin master

Once you login to AWS and set up a cloud server, you’ll be pulling code from your GitHub repository.

**EC2 | Intro**

While there are many services out there that help with application deployment, we'll be using Amazon EC2. This service provides easily scalable servers and storage space in the cloud that makes deployment easy. Also, the lowest-tier servers are free.

**Getting Started**

Before we get started, make sure you have signed up for AWS Free Tier. If you haven't, go to this link <http://aws.amazon.com/free/>and sign up. AWS requires you to provide a Credit Card during sign up, but don't worry; AWS will only charge you if you purchase non-free services. AWS will not charge you upon signing up.

**Road Map**

1. **Launch an Amazon EC2 Instance**We are going to be renting some space on a computer owned by Amazon. We have been using our own computer, localhost, to host our applications for us. Our computer was not designed to be a server, it was designed to be a client. Amazon has some powerful computers that can run multiple high traffic applications so we will be renting a small corner of a very large and powerful server computer. Note: In this chapter, we will demonstrate how to run a free instance. Amazon EC2 offers up to 1 year of free use so that programmers like us can enjoy the free service.
2. **Connect to our EC2 instance**Now that we've rented our server space, how do we access it? What do we mean by access? It's just like when you open up your terminal and access the files in your own computer. Fortunately, we can use our own terminal to connect to the computer that we bought. It is as if we are accessing the terminal from that computer.
3. **Installation**Your EC2 instance has only the operating system installed. In order to get our project up and running we'll have to install the necessary software. We will replicate the development environment from your computer on the server so that the application will run remotely just as it does locally.
4. **Point**We are going to register our domain name to point to the IP address of this computer that we bought from Amazon. Now whenever someone requests our domain name, the request will look at the address book, and know which IP address to go to.

**Step 3: Creating an EC2 server instance**

*Note: You’ll need an AWS account, which you can sign up for*[*here*](http://aws.amazon.com/)*. It’s free for a year, so long as you don’t have more than 1 (free-tier) instance up at a time!*

1. Login to AWS Console at <https://aws.amazon.com/>
2. Once you have logged in you will see your dashboard page: 
3. Launch a new instance from the EC2 Dashboard, as shown below: 
4. Select *Ubuntu Server 16.04* option. 
5. Select *t2.micro* option and click *Review and Launch* 

**Security settings**

1. Click the *Edit security groups* link in the lower right corner 
2. SSH option should be there already. Reset SSH source from the dropdown menu to MyIP. This is the ip address of your current location. If you go home, for example, you will have to set this again to get it to be your home ip. 
3. Click the add a rule button twice to add HTTP and HTTPS, source set to *Anywhere*, and then click *Review and Launch:* 
4. You’ll be asked to create a key file. This is what will let us connect and control the server from our local machine. Name your pem key whatever makes the most sense to you as shown in the next step. Give it a generic name, not the name of your project, as we will be re-using this instance. The key will automatically be saved to your downloads folder when you click Download Key Pair, but you will want to move it. See the next item for more information on this critical step. 
5. This next part is very important! Put your pem key in a file that has no chance of *EVER* being pushed to github. You should not send this file via email, or in any other way make it publicly available: 
6. After launching your instance, you will see a rather confusing screen with some information, as shown below. In order to move on, scroll to the bottom of the page and confirm that you would like to view your instance. 
7. Once you have several instances running, you will want to be able to identify what your different instances are for. We have the option of naming our instance, so let’s do so now by clicking on our instance’s name column as shown. 

**Step 4: Connecting to your remote server**

Back in your terminal, cd to the folder that holds the key file you just downloaded.

> cd /projects/AWS

Now we’re ready to use our .pem file to connect to the AWS instance! In your AWS console, click connect and use the supplied code in your terminal (PC users: use a bash terminal to do this).

Back in your AWS console click the connect button at the top of your screen here:



A popup will appear with instructions on how to connect. The commands in red boxes are the ones you should run.

**If you are a Mac user:**

Run the chmod command in your terminal.

**If you are a Windows user:**

SSH is not built into windows. The easiest solution is to use Git Bash or another bash terminal. The other solution requires the installation of other 3rd-party software, an SSH client. If you wish to do so, we recommend PuTTY.

**Everyone:**

Copy and paste the line starting with ssh (below) and paste the text into your terminal.



You might have to type yes and wait for a few seconds or up to a minute before you are connected, but if all goes well, you should be on your Ubuntu cloud server. Your terminal should show something like this in the far left of your prompt:

ubuntu@54.162.31.253:~$ #Commands you write appear here

**Step 5: Server Configuration**

Now we are going to set up our remote server for deployment. Our server is nothing more than a small allocated space on someone else’s larger computer (in this case, the big computer belongs to Amazon!). That space has an installed operating system, just like your computer. In this case, we are using a distribution of Linux called Ubuntu, version 16.04.

Although we have linux, our new computer is otherwise empty. Let’s change that so we can start building a server capable of providing content that the rest of the world can access. In order to do so, we have to install some key programs first. First, let’s install python, python dev, pip, nginx, and git

In the terminal:

ubuntu@54.162.31.253:~$ sudo apt-get update

ubuntu@54.162.31.253:~$ sudo apt-get install python-pip python-dev nginx git

Now that we’ve installed some packages using apt-get, let’s run update again to make sure apt-get knows we’ve done those installations.

In addition, you’ll use your newly installed pip to install the virtual environment program so that you can now build a brand new virtual environment on your new computer.

ubuntu@54.162.31.253:~$ sudo apt-get update

ubuntu@54.162.31.253:~$ sudo pip install virtualenv

If pip install does not work, try sudo pip3 install virtualenv.

Now you’ll clone your project from GitHub. You’ll type into your terminal something that looks like this:

ubuntu@ip-my-ip:~$ git clone https://github.com/AnnaBNana/django\_courses.git

At the moment, your current folder directory should look something like this.

- ubuntu

- repoName

- apps

- projectName

- ...# other files/folders

Navigate to the project by doing cd myRepoName, and then run ls in your terminal. If you don’t see manage.py as one of the files, *STOP*. Review the setting up GitHub/Git pieces from earlier.

If everything looks good, let’s make that virtual environment, and activate it.

ubuntu@54.162.31.253:~/myRepoName$ virtualenv venv --python=python3

ubuntu@54.162.31.253:~/myRepoName$ source venv/bin/activate

(venv)ubuntu@54.162.31.253:~/myRepoName$ pip install -r requirements.txt

(venv) ubuntu@54.162.31.253:~/myRepoName$ pip install django bcrypt django-extensions

(venv) ubuntu@54.162.31.253:~/myRepoName$ pip install gunicorn

**VIM**

If you have used VIM before, skip to the next tab.

VIM is a terminal based file editor. We will use it to change the necessary files in order to get our project running. In the following instructions, you'll be using the vimcommand to enter the editor. The vim command can be used either to edit existing files or create and open a new blank file. Once you have entered the editor interface, press i to enter INSERT mode. You should see **–INSERT–** at the bottom left corner of your terminal. Now use your arrow keys to move the cursor to where you want to edit and make your changes.

Once you are done, press the esc key to exit INSERT mode. Type a colon to enter the vim command interface. You should now see a colon at the bottom left corner of your terminal. Now, type wq and press return to write (save) and quit.

If you want to quit without saving, type q! after the colon.

If you'd like to save without quitting, type w after the colon.

**Step 6: Editing Settings**

**IMPORTANT: NOTE FOR STEP 6 and BELOW**

**Anywhere you see {{myRepoName}} – replace that whole thing INCLUDING the {{}} with your outer folder name.**

**Anywhere you see {{projectName}} – replace that whole thing INCLUDING the {{}} with the project folder name.**

**Anywhere you see {{yourEC2.public.ip}} – replace that whole thing INCLUDING the {{}} with the public IP address of your newly created server.**

If you named your repo something different from your project, the repo name and project name may be different, but it is ok if they are the same.

Navigate into your main project directory (where settings.py lives). We’re going to use a built-in text editor in the terminal to update the code in settings.py. For example:

(venv) ubuntu@54.162.31.253:~/myRepoName$ cd {{projectName}}

(venv) ubuntu@54.162.31.253:~/myRepoName/projectName$ sudo vim settings.py

You'll need to add a line that allows you to serve static content. You'll also need to modify a couple of lines, as follows:

# Inside settings.py

# modify these lines

DEBUG = False

ALLOWED\_HOSTS = ['{{yourEC2.public.ip}}']

# add the line below to the bottom of the file

STATIC\_ROOT = os.path.join(BASE\_DIR, "static/")

Save your changes and quit.

Run cd .. to get back to the folder that holds manage.py. Make sure your virtual environment is activated!

# Say 'yes' to the dialogue box after typing this next command:

(venv) ubuntu@54.162.31.253:~myRepoName$ python manage.py collectstatic

**Step 7: Gunicorn**

You may remember that Gunicorn is our process manager. Let's test Gunicorn by directing it to our Django project's wsgi.py file, which is the entry point to our application.

(venv) ubuntu@54.162.31.253:~myRepoName$ gunicorn --bind 0.0.0.0:8000 {{projectName}}.wsgi:application

If your Gunicorn process ran correctly, you will see something like the following printed to the terminal:

[2016-12-27 05:45:56 +0000] [8695] [INFO] Starting gunicorn 19.6.0

[2016-12-27 05:45:56 +0000] [8695] [INFO] Listening at: http://0.0.0.0:8000 (8695)

[2016-12-27 05:45:56 +0000] [8695] [INFO] Using worker: sync

[2016-12-27 05:45:56 +0000] [8700] [INFO] Booting worker with pid: 8700

To exit the process ctrl-c

Now, deactivate your virtual environment.

Next, let's set up Gunicorn to run as a service. You'll be using *systemd* as your init system to manage and control aspects of your server including services. The primary advantage of turning Gunicorn into a service is that Gunicorn will start with the server after being rebooted and once configured will just work.

To be able to turn Gunicorn on and off,  we're going to create a *systemd* service file and make some changes.

Our current folder structure should look like this:

~ #home

- ubuntu

- {{repo\_name}}

- {{project\_name}}

- apps

- venv

      - #other files and folders...

Enter:

ubuntu@54.162.31.253:~myRepoName$ sudo vim /etc/systemd/system/gunicorn.service

In the vim text editor copy and paste the following code. Don’t forget to type i before copying and pasting the lines below, otherwise vim may cut off a few characters at the beginning!

[Unit]

Description=gunicorn daemon

After=network.target

[Service]

User=ubuntu

Group=www-data

WorkingDirectory=/home/ubuntu/{{repoName}}

ExecStart=/home/ubuntu/{{repoName}}/venv/bin/gunicorn --workers 3 --bind unix:/home/ubuntu/{{repoName}}/{{projectName}}.sock {{projectName}}.wsgi:application

[Install]

WantedBy=multi-user.target

*REMINDER: myRepoName is the name of the repo you cloned. projectName is the name of the folder that was used when you ran the django-admin startproject command. This folder is sibling to your apps folder.*

Now that our service file has been created, we can enable it so that it starts on boot with these commands.

ubuntu@54.162.31.253:~$ sudo systemctl daemon-reload

ubuntu@54.162.31.253:~$ sudo systemctl start gunicorn

ubuntu@54.162.31.253:~$ sudo systemctl enable gunicorn

*Note:* if any additional changes are made to the gunicorn.service the previous three commands will need to be run in order to sync things up and restart our service.

**Step 8: Nginx**

One final file to edit. From your terminal:

ubuntu@54.162.31.253:~$ sudo vim /etc/nginx/sites-available/{{projectName}}

Add the following to your new document, editing what’s inside curly braces {{}}:

server {

listen 80;

server\_name {{yourEC2.public.ip}};

location = /favicon.ico { access\_log off; log\_not\_found off; }

location /static/ {

root /home/ubuntu/{{myRepoName}};

}

location / {

include proxy\_params;

proxy\_pass http://unix:/home/ubuntu/{{myRepoName}}/{{projectName}}.sock;

}

}

Save and exit.

Now in the terminal, run the following (taking note of the space after {{projectName}}):

ubuntu@54.162.31.253:~$ sudo ln -s /etc/nginx/sites-available/{{projectName}} /etc/nginx/sites-enabled

ubuntu@54.162.31.253:~$ sudo nginx -t

**Step 9 - Finally:**

We will remove the Nginx default site display from directory sites-enabled, by running the following in your terminal.

ubuntu@54.162.31.253:~$ sudo rm /etc/nginx/sites-enabled/default

Now, all that is left to do is restart your Nginx server.

ubuntu@54.162.31.253:~$ sudo service nginx restart

If your server restarted correctly, you will see the new command line, and your app is deployed! Go to the public domain and your app should be there. If you see anything other than your app, review your server file for errors.

**Common errors and how to find them:**

* 502, bad gateway: there is a problem in your code. Hint: any error starting with 5 indicates a server error
* Your Gunicorn process won’t start: Check your .service file; typos and wrong file paths are common mistakes
* Your NGINX restart fails: Check your NGINX file in the sites-available directory. Common problems include typos and forgetting to insert your project name where indicated.
* Make sure the URL requested is correct (example if your root route is /home, make sure you put /home after the IP)

**Step 10: Adding a MySQL server (optional)**

SQLite is great for testing, but it’s not really efficient in the context of real-world use. This may be a bit too much to go into here, but let’s look at a quick summary of why that is, and what we’ll use instead.

Although the developers of SQLite have done much to improve its performance, particularly in version 3, it suffers from some lack of efficient write concurrency. If your site has a lot of traffic a queue begins to form, waiting for write access to the database. Before long, your response speed will slow to a crawl. This happens only on high-traffic sites, however.

MySQL databases, on the other hand, are incredibly fast, and very good at performing multiple operations concurrently. In addition, MySQL can store an incredibly large amount of data, and so is said to scale well.

This might never be a consideration for small to medium sized projects, but is key information in the real world. Very soon you may be working for a company that handles a large volume of requests, and it is important to know why depending on a SQLite database alone is not a practical solution for enterprise or large startups.

If you’d like to learn how to add a MySQL database to the app we just deployed, read on. It’s not as hard as you might think, thanks to Django migrations!

First, we’ll need to install everything necessary to run MySQL from our deployment machine.

ubuntu@54.162.31.253:~$ sudo apt-get install libmysqlclient-dev

Let’s install MySQL-server, then we’ll create a MySQL database and database user for our Django application. After running the command below, set your MySQL password for root as root.

ubuntu@54.162.31.253:~$ sudo apt-get install mysql-server

Then let’s just make sure that we were able to install it correctly. First, we’re going to switch over to root user.

ubuntu@54.162.31.253:~$ sudo su

The su command stands for “switch user”. If we don’t specify a user, Linux/Unix defaults to root user, which is the administrative user. This command provides access to everything in your system and therefore can be dangerous if you aren’t familiar with the effect of the commands you’re using. Be careful not to type any commands other than the following until we exit root user a little later.

The command below will open up MySQL console. You have not interacted with MySQL in the command line before because we used MySQL workbench to interact instead. We may not have a GUI available, but it’s pretty easy to interact with MySQL in the command line.

ubuntu@54.162.31.253:~$ mysql -u root -p

You’ll see the MySQL prompt where we can set up our database.

Let’s create the database for our project. You can call it whatever you want but we recommend giving the name of your project. Note, every command must end with a semi-colon. Make sure to check for them if you have any errors or your commands do not run. Now to create a database on our MySQL server.

CREATE DATABASE {{projectName}};

Exit the MySQL prompt by typing exit;

**Important!!! After exiting the MySQL prompt you MUST type the command**exit**again!**

That’s right, we’re typing exit twice. The first time is to exit the MySQL prompt, the second time is to deactivate the root user. As we warned above, this is a critical step and can result in some problems with installations if we skip it.

Let's make a final installation. If you’re in your outer project directory you must cd into the directory containing your settings.pyfile. If you have followed instructions, you will type:

ubuntu@54.162.31.253:~$ cd {{projectName}}

Activate your virtual environment: source venv/bin/activate. Now we just have to install a pip module inside our virtual environment to help connect our python code to our MySQL database:

(venv) ubuntu@54.162.31.253:~$ pip install mysql-python

Now that we have MySQL all set up, we are ready to change some lines in our settings.py document so we can start working with our MySQL database!

> sudo vim settings.py

Change the databases section in settings.py to look like below:

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.mysql',

'NAME': '{{projectName}}',

'USER': 'root',

'PASSWORD': 'root',

'HOST': 'localhost',

'PORT': '3306',

}

}

remember how to exit vim, by pressing esc, :wq

We’re almost done! Now the only thing left to do is to make migrations!

(venv) ubuntu@54.162.31.253:~myRepoName$ python manage.py makemigrations

(venv) ubuntu@54.162.31.253:~myRepoName$ python manage.py migrate

Now just need to restart Nginx:

sudo systemctl restart gunicorn sudo service nginx restart

Now visit your site! You should be finished at this point, with a fully functioning site. Your old data will not show up, but you should be able to perform all operations as you did previously.

**Step 10.01: Reconnecting**

Remember how we said that we would have to change our security settings every time our IP changes? The bad news is that this will happen often. The good news is that it’s easy to change those settings, if you know where to look.

1. In your AWS console, with your instance selected, scroll down to view some options. Next to security groups, you will see launch-wizard. Click it! 
2. Now you just have to update the IP connected to the instance. In the next window you will see something like this at the bottom of your screen. Click the inbound tab, and then select edit.  
   
3. Now, all that is left to do is let AWS automatically change our IP to the new one. Do this by selecting the dropdown in the SSH row, under source, and select MyIP (it is already selected, but doing so again will refresh your IP to the current one). Once this is done, click save. You’re ready to SSH into your instance again! 