**DOCKER PROJECT1**

**Deploy a python base ecommerce website on an ubuntu server with nginx and gunicorn**

Consider you working in the middleware team and the developer team needs a server to run and test an ecommerce application. Go ahead and use the confluence documentation to install a test environment for the dev team.

Set Up Django app with Postgresql, Nginx, and Gunicorn on Ubuntu

**Preliminary note**

in this tutorial we will deploy an ecommerce python base website on a virtual ubuntu server.

**Most important prerequisites for this project**

* Ubuntu
* nginx
* gunicorn
* Django
* PostgreSQL

**Step 1: Creating and configuring our ubuntu virtual machine.**

* Open your gitbash and move to the home directory

cd ~

* Create a new directory called **myecommerapp** and navigate into it

mkdir myecommerceapp

cd myecommerceapp

* Now create a new file **Vagrantfile** in the **myecommerceapp** folder

vi Vagrantfile

* Copy and paste the code below inside, and save.

# -\*- mode: ruby -\*-

# vi: set ft=ruby :

VAGRANTFILE\_API\_VERSION = "2"

Vagrant.configure(VAGRANTFILE\_API\_VERSION) do|config|

config.ssh.insert\_key = false

config.vm.provider :virtualbox do|vb|

vb.customize ["modifyvm", :id, "--memory", "2048"]

end

#Web Server

config.vm.define "website-server" do|web|

web.vm.hostname = "website-server"

web.vm.box = "bento/ubuntu-18.04"

web.vm.network "private\_network", ip: "192.168.43.14"

end

config.vm.box\_check\_update = false

config.vbguest.auto\_update = false

end

* Now start the virtual server:

vagrant up

We have our server running so let's go deploy our e-commerce website

Naming the host

Ssh to your ubuntu server. The default password is vagrant.

vagrant ssh website-server

After entering the correct password, you should be logged in as root to the web server. Now you are ready to update the server, the software and its dependencies and also install python-pip for the installation of python base debendencies.

In the Webserver, switch to the root user then run the commands:

sudo su

apt-get update && apt-get upgrade

apt-get -y install python3-pip nginx

pip3 install --upgrade pip

The next step would be to give the web server a **hostname** and add the **IP address** and the given name in the host file located in the /etc/ path of your linux distribution.

We are going to give our host server a name of **website** and the ip of the host is **192.168.43.14**

cd ~

hostnamectl set-hostname website

vi /etc/hosts

In the file that you just opened, add the hostname and the api address just below the local host as shown below

127.0.0.1 localhost

192.168.43.14 website

# The following lines are desirable for IPv6 capable hosts

::1 localhost ip6-localhost ip6-loopback

ff02::1 ip6-allnodes

ff02::2 ip6-allrouters

Save the file and exit.

Installing the firewall

The next step should be to install the firewall first and then configure it. Here we allow outgoing and deny the incoming traffic.

sudo apt-get install ufw

sudo ufw default allow outgoing

sudo ufw default deny incoming

Now it's important to allow SSH in the rules before we enable them. For test purposes we open port **8000** in the rules as well. Finally we enable the rules of the firewall ufw, an acronym that stands for **Uncomplicated firewall.** The final command status displays the rules currently in force.

// run the command to allow ssh connection

sudo ufw allow ssh

// here we configure the fire wall to allow port 8000

sudo ufw allow 8000

// here we are enabling the fire wall

sudo ufw enable

// here we are getting the status of the fire wall to confirm that it is active.

sudo ufw status

Status: active

To Action From

-- ------ ----

22/tcp ALLOW Anywhere

8000 ALLOW Anywhere

22/tcp (v6) ALLOW Anywhere (v6)

8000 (v6) ALLOW Anywhere (v6)

Installing the PostgreSQL database on Ubuntu

The next thing we will do is to install PostgreSQL as our database and latter connect it to our application

sudo apt-get install postgresql postgresql-contrib

Next let's set the password and username of our database then switch to the user. I will use a simple password **utrains** but you can use any one you are confortable with just make sure you take note of it.

sudo passwd postgres

 Switch to the posgres user account

sudo -u postgres psql

On the PostgreSQL terminal, we first create the database for the website and a user who performs the database queries . You have to grant this very user all privileges. This sensitive data such as database, user name or password must of course be stored in the database area of **settings.py** or the corresponding environment variables. With the last command \q the terminal can be left again.

As the posgres user: postgres-# run the commands:

postgres-# CREATE DATABASE website;

postgres-# CREATE USER webuser WITH PASSWORD 'utrainsdb';

postgres-# GRANT ALL PRIVILEGES ON DATABASE website to webuser;

postgres-# \q

Finally, we restart the PostgreSQL server back inside the virtual environment. Using the second command is useful if you want to examine the system status of PostgreSQL.

sudo service postgresql restart

\* Restarting PostgreSQL 9.3 database server [ OK ]

sudo service postgresql status

9.3/main (port 5432): online

**Step 2: Clone the website code from GitHub, configuring the database access and installing the application base dependencies.**

In this stage we will clone our application from the website repository and then install the application based dependencies. We will also configure the application to access our **postgresql** database.

Clone the website code from GitHub.

Now that we have completed the Linux configuration, it is time to clone our website code from our github repository

We are going to use git, so first we will install git, then use it to clone the repository.

apt-get -y install git

git clone https://github.com/tatahnoellimnyuy/eccommerce.git

ls

If everything worked fine, the project folder should be listed inside the user's home directory.

Adding our postgres configurations to the python app

For our application to connect to postgresql we need to add the user configurations in the **settings.py** of our app.

vi eccommerce/demo/settings.py

scroll down to DATABASES and edit the configuration as shown below.

A picture containing text, screenshot, font

Description automatically generated

Creating a virtual environment to install the python website dependencies

Now let's create a virtual environment where we will install the dependencies of our application. We will need to install virtualenv used for the creation of python virtual environments

cd ~

pip3 install virtualenv

Now we can create the virtual environment. Inside our ecommerce project folder there will be generated a new folder that manages the virtual environment. The name of the folder is in our case venv.

virtualenv eccommerce/venv

cd eccommerce

source venv/bin/activate

After switching to the Django project folder, the virtual environment can now be activated.

Now, if the prompt brackets **(venv)** are displayed at the beginning, it´s a good sign, that the virtual environment was successfully activated. Now it's time to install all dependencies.

pip3 install -r requirements.txt

Collecting static files of our Ecommerce app

Now run the command below to collect all the static folders and files and also the migrations into the database (in the Web Server (Virtual Environment))

python manage.py collectstatic

python manage.py makemigrations

python manage.py migrate

Now you can start the web server on port **8000** like in the development mode. As you can remember we had set up the firewall accordingly to be open port **8000**.

python manage.py runserver 0.0.0.0:8000

After entering the IP address and port in the address bar of the browser (<http://192.168.43.14:8000/>) the website should render properly. You can see for yourself by testing the functionality of the website.

A screenshot of a website

Description automatically generated with low confidence

The website was running using the **dev server** but for best practice we need to use a more reliable server. **Note**: The dev server is vulnerable to attacks and it’s built just for development purpose.

To stop the **dev sever**, press **CTRL+ C** . In case you are logged out of the server, just ssh back into the server and change to the root user again.

Step 3: Configure reverse proxy server using nginx and gunicorn.

Here we will configure nginx and gunicorn to serve our application on port 80.

Test Gunicorn

Next, you will need to test whether the Gunicorn can serve the Django or not. You can start the Gunicorn server with the following command. you must be in the ecommerce directory before running this command and your virtual enviroment should be activated.

gunicorn --bind 0.0.0.0:8000 demo.wsgi

If everything is fine, you should get the following output:

[2021-06-22 11:20:02 +0000] [11820] [INFO] Starting gunicorn 20.1.0

[2021-06-22 11:20:02 +0000] [11820] [INFO] Listening at: http://0.0.0.0:8000 (11820)

[2021-06-22 11:20:02 +0000] [11820] [INFO] Using worker: sync

[2021-06-22 11:20:02 +0000] [11822] [INFO] Booting worker with pid: 11822

Press **CTRL + C** to stop the Gunicorn server.

Next, deactivate the Python virtual environment with the following command:

deactivate

Create a Systemd Service File for Gunicorn

It is a good idea to create a systemd service file for the Gunicorn to start and stop the Django application server.

To do so, create a socket file with the following command:

vi /etc/systemd/system/gunicorn.socket

Add the following lines to the file:

[Unit]

Description=gunicorn socket

[Socket]

ListenStream=/run/gunicorn.sock

[Install]

WantedBy=sockets.target

Save and close the file then create a service file for Gunicorn:

vi /etc/systemd/system/gunicorn.service

Add the following lines that match your Django project path:

[Unit]

Description=gunicorn daemon

Requires=gunicorn.socket

After=network.target

[Service]

User=root

Group=www-data

WorkingDirectory=/root/eccommerce

ExecStart=/root/eccommerce/venv/bin/gunicorn --workers 3 --bind unix:/run/gunicorn.sock demo.wsgi:application

[Install]

WantedBy=multi-user.target

Save and close the file then set proper permission to the Django project directory:

chown -R www-data:root ~/eccommerce

Next, reload the systemd daemon with the following command:

systemctl daemon-reload

Next, start the Gunicorn service and enable it to start at system reboot:

systemctsystemctl start gunicorn.socket

systemctsystemctl enable gunicorn.socket

 To check the status of the Gunicorn, run the command below:

systemctl status gunicorn.socket

You should get the following output:

● gunicorn.socket - gunicorn socket

Loaded: loaded (/etc/systemd/system/gunicorn.socket; enabled; vendor preset: enabled)

Active: active (running) since Tue 2021-06-22 12:05:05 UTC; 3min 7s ago Triggers: ● gunicorn.service

Listen: /run/gunicorn.sock (Stream)

CGroup: /system.slice/gunicorn.socket

Jun 22 12:05:05 django systemd[1]: Listening on gunicorn socket.

Configure Nginx as a Reverse Proxy to Gunicorn Application

Next, you will need to configure Nginx as a reverse proxy to serve the Gunicorn application server.

To do so, create an Nginx configuration file:

vi /etc/nginx/conf.d/django.conf

Add the following lines:

server {

listen 80;

server\_name 192.168.43.14;

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

location /static\_root/ {

root /root/eccommerce;

}

location / {

include proxy\_params;

proxy\_pass http://unix:/run/gunicorn.sock;

}

}

Save and close the file then verify the Nginx for any configuration error:

nginx -t

Output:

nginx: the configuration file /etc/nginx/nginx.conf syntax is ok

nginx: configuration file /etc/nginx/nginx.conf test is successful

Before we start the Nginx server again, we close port 8000 in the firewall, which we opened earlier for testing purposes. We also allow http traffic.

sudo ufw delete allow 8000

[...]

sudo ufw allow http/tcp

[...]

Finally, restart the Nginx service to apply the changes.

systemctl restart nginx

You should now be able to go to your server’s domain or IP address to view your application. 192.168.43.14

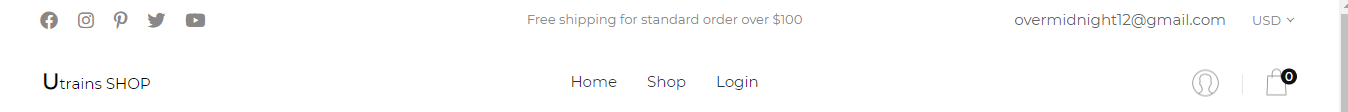
A picture containing clothing, screenshot, person, fashion design

Description automatically generated

**Creating a user and logging in**

Now lets test our application by signing up a user and then logging in as the user.

On the navigation bar hover on login and click on **Sign up**



Enter your information and click on sign up

A screenshot of a login form

Description automatically generated with medium confidence

Go back to your home page and sign up

**Docker project2**

**Containerizing and application with docker (case study python django application)**

Containerizing and application with docker.(case study python django application)

Your company has an application running on a virtual machine but maintaining this application is very stressful.You are the team leader in charge of proposing a better solution. After doing some research you decided together with your team to containerize the application using Docker . To containerize this application you needed to go through multiple tests steps.

Step1:

* Organised a meeting with the developers :

The aim of this meeting is to understand the functioning of the application. During the meeting the developers had to answer the following questions.

1. What operating system the applications running on
2. The technology used to build the application
3. Is the application having a database? And if yes what type of database.
4. Are there any dependencies needed to run the application ? If there are any extra dependencies what are they.
5. The server (reverse proxy)  used to deploy the application on the server
6. The port the application is running on
7. And the command needed to start the application.

After the meeting with the developers you had the following answers.

1. The application is running on an ubuntu server
2. The technology used to build the application is python Django
3. The Database used is postgresql. ( in this case you agreed with the developers that for the step 1 test case you won’t use the postgresql but instead you will use the Django built in sqlite database. So you had to organise a meeting to do the modifications
4. The developers had to create a requirements.txt file and add in the application code directory
5. The application is deployed using using nginx and gunicorn so the developers provided you with the config files
6. The application is running on port 8020
7. You concluded with the developers to create a bash shell script to start the application when executed.

* Once you had the answers to your questions you went ahead to containerize this application. Follow the steps [here](https://dataservicegroup.atlassian.net/l/cp/D3Pf6jV1)

**NOTE**: These are the steps to Containerize any application with Docker.

Prerequisites:

In this tutorial we assumed that

* You have the website-server machine up and runing
* you have docker installed in the server

First login via ssh to your ubuntu server. the default password is vagrant.

vagrant ssh website-server

After entering the correct password, you should be logged in as root to the web server (gain root access with sudo -i ). Now you are ready to containerize the application. The first thing to do is to clone the application from the github repository

Clone the website code from GitHub.

We are going to use git, so first we will install git if you dont have it installed already, then use it to clone the repository.

apt-get -y install git

git clone https://github.com/tatahnoellimnyuy/eccommerce.git

If everything worked fine, the project folder should be listed inside the user's home directory.

// if the clone is successfull you will get see the eccommerce folder in the home directory

ls

Next lets modify our database to use the default database of django. Here we are not using the postgres database because this is still the test phase and we want to run everything in a single container.

So in other to modify the database for the test phase you will need to communicate with the developer.

Note that this step will vary depending on the the application infrastructure so always communicate with the developers.

Modifying the database

Here we will use the django sqlite default database

vi eccommerce/demo/settings.py

Scroll down to DATABASES and uncomment the sqlite database and comment the postgresql as shown below

The application is set and is ready to be containerize. We are going to use docker to containerize this application. The steps are as follows:

**Part 1: Writing the Dockerfile**

1. Choose a base Image
2. Install the necessary packages.
3. Add your custom files
4. Define which user will (or can) run your container. ...
5. Define the exposed ports. ...
6. Define the entrypoint.

**Part 2: Buid and run the container**

Dockerizing the Application

Lets set up the project to run the application in Docker using a more robust web server that is built to handle production levels of traffic:

* **Gunicorn**: [Gunicorn](http://gunicorn.org/) is an HTTP server for Python. This web server is robust and built to handle production levels of traffic, whereas the included development server of Django is more for testing purposes on your local machine only. It will handle all dynamic files.
* **Ngnix**: is a general-purpose HTTP server, we’ll use it as a reverse proxy to serve static files.

On a regular server, setting the application would be hard work; we would need to install and configure Python and Ngnix or apache , then open the appropriate ports in the firewall. Docker saves us all this work by creating a single image with all the files and services configured and ready to use. The image we’ll create can run on any system running Docker.

To dockerize our application we need to write a Dockerfile this file will be use to build our image.

Part 1: Writing the Dockerfile

The next stage is to add a Dockerfile to your project. This will allow Docker to build the image it will execute on the Docker Machine you just created. Writing a Dockerfile is rather straightforward and has many elements that can be reused and/or found on the web. Docker provides a lot of the functions that you will require to build your image. If you need to do something more custom on your project, Dockerfiles are flexible enough for you to do so.

**Step 1: choosing the Base the Image**

You can start from a Base OS and install everything by yourself. But there is no need to create a brand new image. Just pick one of the public images with all the functions and databases that you need.

Choose an image based on the used technology, such as:

* [Ruby](https://hub.docker.com/_/ruby/)
* [Node](https://hub.docker.com/_/node/)
* [Java](https://hub.docker.com/_/java/)
* [Python](https://hub.docker.com/_/python/)

In our case the application is base on on python so the preferred base image will be a python base image.

The structure of a Dockerfile can be considered a series of instructions on how to build your container/image. For example, the vast majority of Dockerfiles will begin by referencing a base image provided by Docker. Typically, this will be a plain vanilla image of the latest Ubuntu release or other Linux OS of choice. From there, you can set up directory structures, environment variables, download dependencies, and many other standard system tasks before finally executing the process which will run your web application.

Start the Dockerfile by creating an empty file named Dockerfile in the root of your project.

cd eccommerce

touch Dockerfile

vi Dockerfile

Then, add the first line to the Dockerfile that instructs which base image to build upon.

FROM python:3.9-buster

**Note:** You can create your own base image and use that for your containers, which can be beneficial in a department with many teams wanting to deploy their applications in the same way.

It’s worth noting that we are using a base image that has been created specifically to handle Python 3.9 applications and a set of instructions that will run automatically before the rest of your Dockerfile.

We could use an Ubuntu base image but it will require that we install python but using a python base image will save us from all this stress.

**Step 2: installing th requirements**

At this stage you will configure the docker file to install the requirements needed to run the application.

In our case we will install **nginx** and the requirements of our application in the **requirements.txt** file.

Note that the **requirements.txt** file is generaly created by the developer.

RUN apt-get update && apt-get install nginx vim -y --no-install-recommends

COPY requirements.txt

RUN pip install -r requirements.txt

**Step 3: copy the required files**

In this stage we copy all the the neccesary files our applicationtion will need to run into our container.

In the case of our aplication, we need the following:

* Nginx configuration file
* The bash file to start our application
* The application code itself.

Before copying the files you will need to create the nginx configuration file, the startup bash shell file and the .**dockerignore** file.

Note that this solution is for this particular case and might vary depending on the technology use to build the application.

Create a new file called nginx.default.

touch nginx.default

Open the file and paste the code below in it

server {

listen 8020;

server\_name 0.0.0.0;

location / {

proxy\_pass http://localhost:8010;

proxy\_set\_header Host $host;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

}

location /static\_root {

root /opt/app/ecomerce;

}

}

This will be our configuration for **nginx**. We’ll listen on port 8020, serve the static files from the /opt/app/ecommerce/static\_root directory and forward the rest of connections to port 8010, where **Gunicorn** will be listening. In other to write this file, during the meeting with the developer he explained to you how the nginx configuration file works. Not that nginx default port is port 80 but you had to change that with the help of the developer.

Create a server startup script called start-server.sh and paste the bash shell script below inside . This is a Bash script that starts **Gunicorn** and **Ngnix**:

touch start-server.sh

chmod +x start-server.sh

#!/usr/bin/env bash

# start-server.sh

if [ -n "$DJANGO\_SUPERUSER\_USERNAME" ] && [ -n "$DJANGO\_SUPERUSER\_PASSWORD" ] ; then

(cd ecommerce; python manage.py createsuperuser --no-input)

fi

(cd ecommerce; gunicorn demo.wsgi --user www-data --bind 0.0.0.0:8010 --workers 3) &

nginx -g "daemon off;"

In the above script you passed the gunicorn command with the first argument of demo.wsgi. This is a reference to the wsgi file Django generated for us and is a Web Server Gateway Interface file which is the Python standard for web applications and servers.

You then pass two flags to the command, bind to attach the running server to port 8020, which you will use to communicate with the running web server via HTTP. Finally, you specify workers which are the number of threads that will handle the requests coming into your application. Gunicorn recommends this value to be set at (2 x $num\_cores) + 1.

Make the script executable:

chmod 755 start-server.sh

Now let’s create the .**dockerignore** file to specify the files we need to ignore. Create a file name .**dockerignore** and specify the various files to ignore when copying to the docker container.

# Ignore Dockerfile

Dockerfile

# Ignoring the virtual environment

venv

The above .**dockerignorefile** defines the files and folders that shouldn’t be copied to the container.

After creating all these files you can then copy them into our container using the copy command.

. . .

COPY nginx.default /etc/nginx/sites-available/default

RUN ln -sf /dev/stdout /var/log/nginx/access.log \

&& ln -sf /dev/stderr /var/log/nginx/error.log

. . .

It’s time to copy the source files and scripts inside the container. We can use the COPY command to copy files and the RUN command to execute programs on build time.

. . .

RUN mkdir -p /opt/app

RUN mkdir -p /opt/app/pip\_cache

RUN mkdir -p /opt/app/ecommerce

COPY requirements.txt start-server.sh /opt/app/

RUN pip install -r requirements.txt

COPY . /opt/app/ecommerce/

WORKDIR /opt/app

RUN python ecommerce/manage.py collectstatic

**Step 4: Define which user owns the app directory in the docker container.**

Here we will set the **www-data** as the owner of the applications directory in the container.

Note that when using any reverse proxy, the **www-data** is always the owner of the application directory.

You must have noticed that we choosed our working directory to be the **/opt/app** directory. This explains why all our folders are created in this directory and our files copied there.

RUN chown -R www-data:www-data /opt/app

. . .

**Step 5: Expose the ports**

The server will run on port 8020. Therefore, your container must be set up to allow access to this port so that you can communicate to your running server over HTTP. To do this, use the EXPOSE directive to make the port available:

. . .

EXPOSE 8020

STOPSIGNAL SIGTERM

CMD ["/opt/app/start-server.sh"]

**Step 6: Define the Entrypoint**

The entry point is the set of commands use to start your container which may vary depending on what you are containerizing. A better way is to create a “*docker-entrypoint.sh*” script where you can hook things like configuration using environment variables. we created a start-server.sh script earlier, this script will be executed to start this particular container. This will leave your web server running on port 8020 waiting to take requests over HTTP. You can execute this script using the CMD directive.

CMD ["/opt/app/start-server.sh"]

With all this in place, your final Dockerfile should look something like this:

# Dockerfile

FROM python:3.9-buster

# install nginx

RUN apt-get update && apt-get install nginx vim -y --no-install-recommends

COPY nginx.default /etc/nginx/sites-available/default

RUN ln -sf /dev/stdout /var/log/nginx/access.log \

&& ln -sf /dev/stderr /var/log/nginx/error.log

# copy source and install dependencies

RUN mkdir -p /opt/app

RUN mkdir -p /opt/app/ecommerce

COPY requirements.txt start-server.sh /opt/app/

COPY . /opt/app/ecommerce/

WORKDIR /opt/app/ecommerce/

RUN pip install -r requirements.txt

RUN python manage.py collectstatic

RUN python manage.py makemigrations

RUN python manage.py migrate

RUN chown -R www-data:www-data /opt/app

# start server

EXPOSE 8020

STOPSIGNAL SIGTERM

CMD ["/opt/app/start-server.sh"]

You are now ready to build the container image, and then run it to see it all working together.

Part 2: Building and Running the Container

Building the container is very straight forward once you have Docker on your system. The following command will look for your Dockerfile and download all the necessary layers required to get your container image running. Afterward, it will run the instructions in the Dockerfile and leave you with a container that is ready to start.

To build your container, you will use the docker build command and provide a tag or a name for the container, so you can reference it later when you want to run it. The final part of the command tells Docker which directory to build from.

$ docker build -t django-project.

In the output, you can see Docker processing each one of your commands before outputting that the build of the container is complete. It will give you a unique ID for the container, which can also be used in commands alongside the tag.

The final step is to run the container you have just built using Docker:

$ docker run -it -p 8020:8020 -d django-project

The command tells Docker to run the container and forward the exposed port 8020 to port 8020 on your local machine. The **-d** command runs django in the background.

After you run this command, you should be able to visit [http://192.168.43.14:8020](http://localhost:8020/) and in your browser to access the application.

In other to use **postgresql** as our database, we will need to use a **multicontainer** system thats where **docker-compose** comes in.