Containarize an Ecommerce app built with python django

NOTE: The steps here are the steps to Containerize any application with Docker.

in this tutorial we assumed that you have docker installed in the server so we will go straight to the steps.

First lets ssh to our server.

vagrant ssh to your ubuntu server. the default password is vagrant.

```
# Local Machine
$ vagrant ssh website-server
```

After entering the correct password, you should be logged in as root to the web server. Now you are ready 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 done installing docker , it is time to clone our website code from our github repository

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.

```
root@website-server:~# apt-get -y install git
root@website-server:~# 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
root@website-server:~# ls
eccommerce
```

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. in the next stage we will create 3 containers. that is the app container, the nginx contianer and the postgresql container.

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

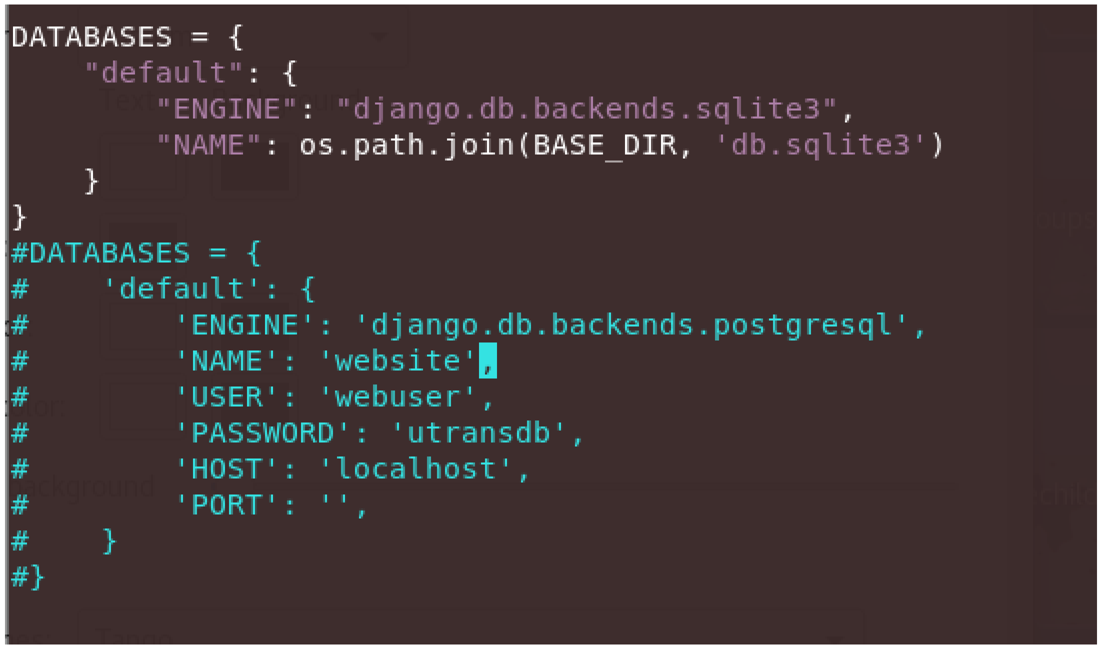
note that this step will vary depending on the the application infrastructure so always communicate with the developers .

Modifying the database

Here we will use the django sqlite defualt database

```
root@website-server:~# vi eccommerce/demo/settings.py
```

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



```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.sqlite3",
        "NAME": os.path.join(BASE_DIR, 'db.sqlite3')
    }
}
#DATABASES = {
#    'default': {
#        'ENGINE': 'django.db.backends.postgresql',
#        'NAME': 'website',
#        'USER': 'webuser',
#        'PASSWORD': 'utransdb',
#        'HOST': 'localhost',
#        'PORT': '',
#    }
#}
```

The application is set and is ready to containerize. we are going to use docker to containerize this application. we will follow the following steps to containarize our application.

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.
7. Build 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 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.

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.

Step1: choosing the Base the Imagine

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
- Node
- Java
- Python

In our case the application is base on on python so the prefared 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. Then, add the first line to the `Dockerfile` that instructs which base image to build upon. 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.

We'll create the Dockerfile in the root of our project:

```
cd eccommerce
touch Dockerfile
```

Create the Dockerfile:

`vi Dockerfile`

```
FROM python:3.9-buster
```

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`.

Step2: 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 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
```

Step3: 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,and the startup bash 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

`vi nginx.default`

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:

```
# nginx.default

server {
    listen 8020;
    server_name example.org;

    location / {
        proxy_pass http://127.0.0.1:8010;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
    location /static_root {
        root /opt/app/ecomerce;
    }
}
```

Create a server startup script called `start-server.sh` and past the bash scipt below inside . This is a Bash script that starts Gunicorn and Ngnix:

```
#!/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;"
```

You then pass 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
```

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
```

Step4: define which user can run the container

here you will define the users in your contianer. in our case here ww-data is the owner of our app folder.

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

. . .
```

Step5: 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"]
```

Step6: Define the Entrypoint

The entry point is the command 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
RUN pip install -r requirements.txt
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.

Step7: 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:

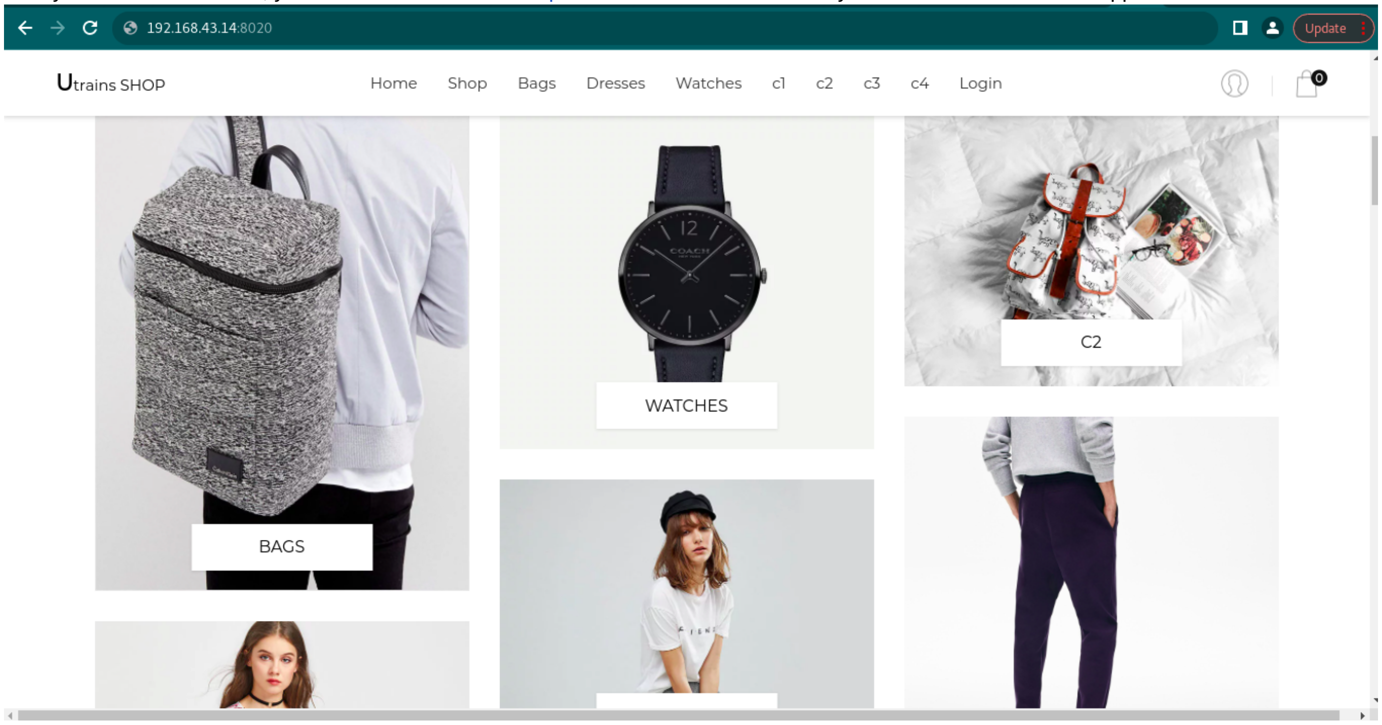
```
$ docker run -it -p 8020:8020 -d django-project
```

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.

now allow port 8020

```
sudo ufw allow 8020
```

After you run this command, you should be able to visit http://192.168.43.14:8020 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.