## Introduction

In this guide, you will build a Python application using the Bottle microframework on Ubuntu 18.04 or later. The bulk of this article will be about how to set up the [uWSGI application server](http://uwsgi-docs.readthedocs.io/en/latest/) and how to launch the application and configure [Nginx](https://www.nginx.com/) to act as a front-end reverse proxy.

INSTALLING UBUNTU UNDER WINDOWS 10/11

Open **cmd** and run: wsl --install

This command will enable the required optional components, download the latest Linux kernel, set WSL 2 as your default.

The above command only works if WSL is not installed at all, if you run wsl --install and see the WSL help text, please try running wsl --list --online to see a list of available distros and then run:

wsl --install -d <DistroName> to install a distro.

This will install and run <DistroName> distribution, for example Ubuntu.

CONFIGURING UBUNTU FOR AN APPLICATION DEVELOPMENT

NOTE: about network access between Windows host and Ubuntu VM:

If you are building a networking app (for example an app running on a NodeJS or SQL server) in your Linux distribution, you can access it from a Windows app (like your Edge or Chrome internet browser) using **localhost** (just like you normally would).

However, if you are running an older version of Windows (Build 18945 or less), you will need to get the IP address of the Linux host VM (or update to the latest Windows version).

## Step 1 — Installing the Components from the Ubuntu Repositories

The first step is to install all of the pieces that you need from the Ubuntu repositories. You’ll install **pip**, the Python package manager, to manage your Python components. You’ll also get the Python development files necessary to build uWSGI.

First, update the local package index:

$ sudo apt update

Then install the packages that will allow you to build your Python environment. These will include python3-pip, along with a few more packages and development tools necessary for a robust programming environment:

$ sudo apt install python3-pip python3-dev python3-setuptools

$ sudo apt install build-essential libssl-dev libpq-dev libpython3-dev

With these packages in place, you can move on to creating a virtual environment for your project.

## Step 2 — Creating a Python Virtual Environment

Next, set up a virtual environment in order to isolate your Bottle application from the other Python files on the system.

Start by installing the python3-venv package, which will install the venv module:

$ sudo apt install python3-venv

Next, make a parent directory for your Bottle project:

mkdir ~/dev/lab5\_app

Then move into the directory after you create it:

cd ~/myproject

Create a virtual environment to store your Bottle project’s Python requirements by running the following:

python3 -m venv myenv

This will install a local copy of Python and **pip** into a directory called **myenv** within your project directory.

Before installing applications within the virtual environment, you need to activate it:

source myprojectenv/bin/activate

Your prompt will change to indicate that you are now operating within the virtual environment. It will read like the following: (myenv) user@host:~/dev/lab5\_app$

## Step 3 — Setting Up a Bottle Application

Now that you are in your virtual environment, you can install Bottle and uWSGI and get started on designing your application.

First, install **wheel** with the local instance of **pip** to ensure that your packages will install even if they are missing wheel archives:

pip install wheel

**Note**: Regardless of which version of Python you are using, when the virtual environment is activated, you should use the **pip** command (not pip3).

Next, install Bottle and uWSGI:

pip install uwsgi bottle

After installation is complete, you can begin using Bottle.

## Step 4 — Setting Up a PostgreSQL connector

For working with PostgreSQL you will also require **psycorg2** python package:

pip install psycopg2

If you have failed with the above command try pip install psycopg2-binary

## Creating a Sample App

Now that you have Bottle available, you can create a simple application. As you may recall, Bottle is a microframework and doesn’t include many of the tools that more full-featured frameworks might. Bottle exists primarily as a module that you can import into your projects to assist you in initializing a web application.

While your application might be more complex, you’ll create your Bottle app in a single file. You can create the file using your favorite text editor. For example, you can use **nano** and name it **lab5.py**:

# Lab-5 sample

# A basic bottle-app skeleton

# This is not a minimal skeleton. This shows you how to serve

# static content, a basic text only front page, and a templated page.

import bottle

app = application = bottle.Bottle()

@app.route('/static/<filename:path>')

def static(filename):

"""

Serve static files

"""

return bottle.static\_file(filename, root='./static')

@app.route('/')

def show\_index():

"""

The front 'index' page

"""

return 'Hello'

@app.route('/page/<page\_name>')

def show\_page(page\_name):

"""

Return a page that has been rendered using a template

"""

return bottle.template('page', page\_name=page\_name)

# While you're testing, you don't want that slash unless the main page

# is being requested. Browsers seem to like adding that. It's not an issue

# when we get to deployment because uwsgi will take care of that for you.

class StripPathMiddleware(object):

"""

Get that slash out of the request

"""

def \_\_init\_\_(self, a):

self.a = a

def \_\_call\_\_(self, e, h):

e['PATH\_INFO'] = e['PATH\_INFO'].rstrip('/')

return self.a(e, h)

# Notice that bottle.run() only happens when you run app.py. It won't run when

# you launch it as an application with uwsgi. This chunk is also the only place

# that we call StripPathMiddleware. If you have no need for development, then you

# can remove the last two chunks of code.

if \_\_name\_\_ == '\_\_main\_\_':

bottle.run(app=StripPathMiddleware(app),

host='localhost',

port=8085,

debug=True)

Try it out!:

python lab5.py

You'll see the application start running. Go to localhost:8085/. Neat, huh?

## Basic App Structure

I tend to continue with a basic folder structure:

/dev/lab5\_app/

plugins/

\_\_init\_\_.py

static/

css/

files/

images/

js/

views/

base.tpl

page.tpl

app.py

Whether I have anything in the directories or not, they exist. It's just how I make sure I keep things consistent across applications.

## The Templating System

Bottle has a bunch of templating options. For now, we're only going to touch the most basic option.

views/page.tpl:

You are visiting {{page\_name}}!

%rebase base

views/base.tpl:

<!DOCTYPE html>

<html lang="en" dir="ltr">

<head>

<title>My Site!</title>

</head>

<body>

<div id="pagebody">

%include

</div>

</body>

</html>

This is obviously very basic, but it will get you started. Check out the [Bottle Docs](http://bottle.readthedocs.org/en/latest/) for more information. The templating options are endless!

Now that you have this done, restart app.py and visit localhost:8085/page/foo. You should be seeing a rather blank looking page that says "You are visiting foo" with the title "My Site!"

https://bottlepy.org/docs/dev/tutorial.html

https://michael.lustfield.net/nginx/bottle-uwsgi-nginx-quickstart

https://www.digitalocean.com/community/tutorials/how-to-serve-flask-applications-with-uswgi-and-nginx-on-ubuntu-18-04

## Step 5 — Configuring uWSGI

Now that we have a very basic bottle application, it's time to fit it into the stack. The built in web server that bottle offers is very slow. It's for development only. Don't ever expect to use it in production. You can now move on to configuring uWSGI.

## Testing uWSGI Serving

Before making more changes, it might be helpful to test that uWSGI can serve your application.

You can do this by passing the name of your entry point to uWSGI. This is constructed by the name of the module (minus the .py extension) plus the name of the callable within the application. In this case the entry point’s name is **lab5:app**

You’ll also specify the socket, so that it will be started on a publicly available interface, as well as the protocol, so that it will use HTTP instead of the uwsgi binary protocol. Use the same port number, **8085**, that you opened earlier:

uwsgi --socket 0.0.0.0:8085 --protocol=http -w lab5:app

Visit your server’s IP address with :8085 appended to the end in your web browser again:

http://your\_server\_ip:8085

You should receive your application’s output again.

When you have confirmed that it’s functioning properly, press **CTRL + C** in your terminal window.

Now that you’re done with your virtual environment, you can deactivate it:

deactivate

Any Python commands will now use the system’s Python environment again.

## Testing uWSGI Serving

Also lets check nginx serving capabilities. To do that lets configure nginx in a simplest way, go to the /etc/nginx folder, rename the nginx.conf to nginx.conf.original and create the new config file:

sudo nano /etc/nginx/nginx.conf

and fill it with this content:

user www-data;

worker\_processes 4;

pid /run/nginx.pid;

events {

worker\_connections 800;

}

http {

server {

location / {

include uwsgi\_params;

uwsgi\_pass 127.0.0.1:8085;

}

}

}

This means “pass every request to the server bound to port 8085 speaking the uwsgi protocol”.

Now we can spawn uWSGI to natively speak the uwsgi protocol:

uwsgi --socket 0.0.0.0:8085 --protocol=uwsgi -w lab5:app

Visit your server’s IP address (localhost, and not appending :8085 this time) in your web browser again, and you should see the output. Also note that no we’ve used --protocol=uwsgi instead of --protocol=http.

## Creating a uWSGI Configuration File

You have tested and verified that uWSGI is able to serve your application, but ultimately you will want something more robust for long-term usage. You can create a uWSGI configuration file with the relevant options for this.

Place that file in your project directory and call it, for example, bottle.ini:

nano ~/dev/lab5\_app/bottle.ini

Inside, you’ll start off with the **[uwsgi]** header so that uWSGI knows to apply the settings. You’ll specify two things: the module itself, by referring to the**lab5.py** file minus the extension, and the callable within the file, **app**:

[uwsgi]

module = lab5:app

Next, tell uWSGI to start up in master mode and spawn five worker processes to serve actual requests:

[uwsgi]

module = lab5:app

master = true

processes = 5

When you were testing, you exposed uWSGI on a network port.

However, you’re going to be using Nginx to handle actual client connections, which will then pass requests to uWSGI. Since these components are operating on the same computer, a Unix socket is preferable because it is faster and more secure. Call the socket uwsgi.sock and place it in the /tmp directory.

Also change the permissions on the socket. This gives the Nginx group ownership of the uWSGI process later on, so make sure the group owner of the socket can read information from it and write to it.

Additionally, clean up the socket when the process stops by adding the vacuum option:

[uwsgi]

module = lab5:app

master = true

processes = 5

socket = /tmp/uwsgi.sock

chmod-socket = 666

vacuum = true

The last thing you’ll do is set the die-on-term option. This can help ensure that the init system and uWSGI have the same assumptions about what each process signal means. Setting this aligns the two system components, implementing the expected behavior:

[uwsgi]

module = wsgi:app

master = true

processes = 5

socket = myproject.sock

chmod-socket = 660

vacuum = true

die-on-term = true

You may have noticed that you did not specify a protocol like you did from the command line. That’s because by default, uWSGI speaks using the uwsgi protocol, a fast binary protocol designed to communicate with other servers. Nginx can speak this protocol natively, so it’s better to use this than to force communication by HTTP.

When you are finished, save and close the file.

And now we need to adjust nginx.conf to use unix socket instead of network port:

user www-data;

worker\_processes 4;

pid /run/nginx.pid;

events {

worker\_connections 800;

}

http {

server {

listen 80;

location / {

include uwsgi\_params;

# uwsgi\_pass 127.0.0.1:8085;

uwsgi\_pass unix:/tmp/uwsgi.socket;

}

}

# note this server section – it is moved from the lab4 to serve REST API requests

# to memcached, but the port is changed to 8089 instead of using the default;

server {

listen 8089;

location / {

set $memcached\_key "$uri";

memcached\_pass 127.0.0.1:11211;

default\_type application/json;

add\_header x-header-memcached true;

}

}

}

That’s all folks! The app is ready to run, restart nginx to apply the configuration:

sudo service nginx restart

and run uwsgi with the following command:

uwsgi --ini bottle.ini

Now open http://localhost in your browser and you should see ‘hello’. Also open <http://localhost/page/foo> and you’ll see ‘You are visiting foo!’.

Also check the REST API functionality which you’ve implemented in lab4, ensure that memcached and postgresql is running:

sudo service postgresql start

sudo service memcached start

Run psql using:

psql or sudo -u postgres psql postgres

Ensure that car\_portal database is exists using **\l** meta-command and execute the following query:

SELECT memcache\_add('/1', 'hello');

Check in your browser http://[localhost:8089/1](http://localhost:8089/1) - you’ll see hello, which means all is OK.

**From that point you can develop the functionality of your app…**