Cloud computing and service oriented Architechture

Tutorial 3: creating a web service using python

In this tutorial, we explain how to create a RESTful web service that intended to act as a data engineering service. The web service is served through the python web framework CherryPy inside a Docker container. Here are the step-by-step instructions on how to achieve that using only 3 files and less than 30 lines of code.

The first step is to setup the development environment by installing Docker, Python 3, and the following Python libraries:

Docker is an application that allows to deploy programs that are run as containers. It was written in the popular Go programming language. This tutorial explains how to install Docker CE on Ubuntu 16.04.

Step 1: Updating all your software

First off, let's make sure that we are using a clean system. Run the apt updater.

sudo apt-get update

Step 2: Set up the repository

Install packages to allow apt to use a repository over HTTPS

sudo apt-get install apt-transport-https ca-certificates curl software-properties-common

Add Docker’s official GPG key

sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

Use the following command to set up the stable repository.

sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"

Step 3: Install Docker CE

sudo apt-get update

sudo apt-get install docker-ce

Step 4: Create a user

If you decide not to run Docker as the root user, you will need to create a non-root user.

Warning: The docker group grants privileges equivalent to the root user.

adduser user

usermod -aG docker user

Restart the Docker service.

systemctl restart docker

Step 5: Test Docker

Run the Docker hello-world container to test if the installation completed successfully.

sudo docker run hello-world

You will see the following output.

Hello from Docker!

This message shows that your installation appears to be working correctly.

Step 6: Configure Docker to start on boot

Lastly, enable Docker to run when your system boots.

sudo systemctl enable docker

Setup your Python environment — Python 3

We are now ready to start configuring our Python development environment for the build. The first step is to install pip , a Python package manager:

$ cd ~

$ wget https://bootstrap.pypa.io/get-pip.py

$ sudo python get-pip.py

I'm a *huge fan* of both virtualenv and virtualenvwrapper. **These Python packages allow you to create** *separate, independent* **Python environments for** *each* **project that you are working on.**

In short, using these packages allows you to solve the *"Project X depends on version 1.x, but Project Y needs 4.x* dilemma. A fantastic side effect of

using Python virtual environments is that you can keep your system Python neat, tidy, and free from clutter.

Again, let me reiterate that it's **standard practice** in the Python community to be leveraging

$ sudo pip install virtualenv virtualenvwrapper

$ sudo rm -rf ~/get-pip.py ~/.cache/pip

Once we have virtualenv and virtualenvwrapper installed, we need to update our ~/.bashrc file to include the following lines at the *bottom* of the file:

 # virtualenv and virtualenvwrapper

export WORKON_HOME=$HOME/.virtualenvs

source /usr/local/bin/virtualenvwrapper.sh

The ~/.bashrc file is simply a shell script that Bash runs whenever you launch a new terminal. You normally use this file to set various configurations. In this case, we are setting an environment variable called WORKON_HOME to point to the directory where our Python virtual environments live. We then load any necessary configurations from virtualenvwrapper .

To update your ~/.bashrc file simply use a standard text editor. I would recommend using nano , vim , or emacs . You can also use graphical editors as well, but if you're just getting started, nano is likely the easiest to operate.

After editing our ~/.bashrc file, we need to reload the changes:

$ source ~/.bashrc

Note: *Calling source on .bashrc only has to be done* **once** *for our current shell session. Anytime we open up a new terminal, the contents of .bashrc will be* **automatically** *executed (including our updates).*

Now that we have installed virtualenv and virtualenvwrapper , the next step is to actually *create* the Python virtual environment — we do this using the mkvirtualenv command.

Creating your Python virtual environment

use this command to create a Python 3 virtual environment:

$ mkvirtualenv ws -p python3

Regardless of which Python command you decide to use, the end result is that we have created a Python virtual environment named cv (short for "computer vision").

You can name this virtual environment whatever you like (and create as many Python virtual environments as you want), but for the time bing, I would suggest sticking with the cv name as that is what I'll be using throughout the rest of this tutorial.

Verifying that you are in the "ws" virtual environment

If you ever reboot your Ubuntu system; log out and log back in; or open up a new terminal, you'll need to use the workon command to re-access your ws virtual environment. An example of the workon command follows:

$ workon ws

To validate that you are in the ws virtual environment, simply examine your command line — *if you see the text (ws) preceding your prompt, then you* **are** *in the ws virtual environment:*

- pandas - for performing aggregation on a dataset
- CherryPy - the web framework for serving the web service

These libraries can be installed using the *pip* command:

>> pip install pandas

>> pip install CherryPy

Let's start by writing a simple data processor *myprocessor.py*:

```
class MyProcessor:

    def run(self, df):

        return df.agg(['mean', 'min', 'max'])
```

This processor computes the mean, minimum and maximum values on all the columns of the input data frame.

In the same directory, we create a web service script *ws.py* that contains the definition of the request handler:

```
import cherrypy

import pandas as pd

import myprocessor

p = myprocessor.MyProcessor()

class MyWebService(object):
```

```
    @cherrypy.expose
    @cherrypy.tools.json_out()
    @cherrypy.tools.json_in()
    def process(self):
        data = cherrypy.request.json
        df = pd.DataFrame(data)
        output = p.run(df)
        return output.to_json()
if __name__ == '__main__':
```

This handler accepts requests directed to the */process* endpoint and expects the input data to be represented as a JSON string in the body of the request. The result from the processing will be returned as a JSON string -conveniently converted using the panda *to_json()* method.

Next come the code for starting the web service in the main section of *ws.py*:

```
if __name__ == '__main__':
    config = {'server.socket_host': '0.0.0.0'}
    cherrypy.config.update(config)
    cherrypy.quickstart(MyWebService())
```

The *myprocessor.py* and *ws.py* are all we need for the web service. We can now start the web service by using this python command:

```
>> python ws.py
```

```
[20/Jun/2019:21:42:59] ENGINE Listening for SIGTERM.
[20/Jun/2019:21:42:59] ENGINE Listening for SIGHUP.
[20/Jun/2019:21:42:59] ENGINE Listening for SIGUSR1.
[20/Jun/2019:21:42:59] ENGINE Bus STARTING
CherryPy Checker:
```

The Application mounted at '' has an empty config.

[20/Jun/2019:21:42:59] ENGINE Started monitor thread 'Autoreloader'.

[20/Jun/2019:21:42:59] ENGINE Serving on http://0.0.0.0:8080

[20/Jun/2019:21:42:59] ENGINE Bus STARTED

Ta da! Here is our web service running in no time.

In a Linux system, we can use *curl* to send this POST request to validate the REST API is working properly:

>> curl -d '{"num1" : [1, 2, 3], "num2":[4, 5, 6]}' -H "Content-Type: application/json" -X POST http://localhost:8080/process

"{\"num1\":{\"mean\":2.0,\"min\":1.0,\"max\":3.0},\"num2\": {\"mean\":5.0,\"min\":4.0,\"max\":6.0}}"

The expected result is a dataset including the mean, minimum and maximum of 2 sets of numbers sent to the web .

Docker container

Next, we need to write a *Dockerfile*—this a text file for setting up the environment which the web service will be running in. If you have not used Docker before, I suggest to at least read the Docker Get Started documentation so you can be familiar with the Docker concepts. The first line in the file states to use *python:3.6.4-slim-jessie* as the base image:

from python:slim-jessie

This is one of the official Python Linux images with a vanilla version of Python 3 pre-installed and there are many other images to choose from the Docker repository, but this is a slim version that is sufficient for running a simple service. Just like how we have to install additional python packages for running the web service locally, the same needs to be done when building the docker image:

RUN pip install pandas

RUN pip install CherryPy

Next, the build process will need to copy the web service files into the default working directory:

COPY myprocessor.py .

COPY ws.py .

The container needs to expose port 8080 to permit access:

EXPOSE 8080

The final statement in the *Dockerfile* defines the command to run when the container starts:

ENTRYPOINT ["python", "ws.py"]

This is what the completed *Dockerfile* looks like and it should reside in the same directory as the python scripts written before:

from python:slim-jessie

RUN pip install pandas

RUN pip install CherryPy

COPY myprocessor.py .

COPY ws.py .

EXPOSE 8080

ENTRYPOINT ["python", "ws.py"]

To execute the docker build command, we stay in the directory where the *Dockerfile* is created. It is good practice to assign docker image with a tag which can be referred to later, so we use the build command with the -t tag to build the image:

>>sudo docker build -t python-ws .

We now have a docker image ready, let’s give this a try:

>>sudo docker run -p 8080:8080 python-ws

This commands starts a container in attached mode, so the Cherrypy logging will appear immediately in the console. Again, the same curl command used for testing before should also work as the host machine (localhost) now forwards requests directed to port 8080 to the running docker container.

We have seen how to build a simple RESTful web service but this service is far from production-ready. Extra security measures should be in place if the web service is hosted in the public cloud, e.g. HTTPS connections, some form of authentication. Example code for how to enable these

security features is available in A hung git up project at https://github.com/hungapl/python-ws. Also, the cherrypy tutorial page https://github.com/hungapl/python-ws provides many examples on how you can customise your web service as well.